Electronics and Computer Science
Faculty of Engineering and Physical Sciences
UNIVERSITY OF SOUTHAMPTON

A group design project report submitted for the award of
MEng Electronic Engineering
MEng Electrical and Electronic Engineering
MEng Electronic Engineering with Computer Systems

Supervisor: Dr. Graeme Bragg
Examiner: Dr. Yasir Noori

Posit for RISC-V

by Robin Ali
Letian Chen
Xiaoan He
Yueyu Guo
Sheng Yang Wong

January 12, 2024

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

A group design project report submitted for the award of MEng Electronic Engineering
MEng Electrical and Electronic Engineering
MEng Electronic Engineering with Computer Systems

by Robin Ali
Letian Chen
Xiaoan He
Yueyu Guo
Sheng Yang Wong

A prototype Posit Processing Unit (PPU), designed to replace the existing IEEE-754 Floating-Point Unit (FPU) within the cv32e40p core using the RISC-V architecture and instruction set, an project undertaken in collaboration with Embecosm. The benefits for this modification are the potential benefits for increased computational accuracy and dynamic range for specific numerical ranges, crucial for certain high-precision arithmetic applications. The project focuses on the implementation of a 32-bit posit format with a 2-bit exponent as per the current standard. Aligning with the RISC-V “F” standard extension operations, the RTL that implements the FPU instruction algorithms were modified to accommodate posit-based arithmetic and non-computational functions for the RISC-V core. The modifications were validated using Universal Verification Methodology (UVM) tests through testbenches. Additionally, a comparative analysis was performed to evaluate the accuracy between the traditional 32-bit IEEE-754 format and the newly introduced 32-bit Posit format, with the 64-bit IEEE-754 standard serving as a secondary reference point. The project remained focused on the replacement of the float operations within the core, running full core tests and was ultimately confined to a simulation environment.

Statement of Originality

We: Robin Ali, Xiaoan He, Letian Chen, Sheng Wong and Yueyu Guo declare and agree with the following statements:

- We have read and understood the ECS Academic Integrity information and the University's Academic Integrity Guidance for Students.
- We are aware that failure to act in accordance with the Regulations Governing Academic Integrity may lead to the imposition of penalties which, for the most serious cases, may include termination of programme.
- We consent to the University copying and distributing any or all of our work in any form and using third parties (who may be based outside the EU/EEA) to verify whether our work contains plagiarised material, and for quality assurance purposes.
- We have acknowledged all sources, and identified any content taken from elsewhere.
- The cv32e40p and core-v verif repositories are used and modified under open-source licensing. The SoftPosit library was also used under open-source licensing. Our final results are made available as well under an open-source license.
- We have carried out all the work between us, and have not helped anyone else.
- The material in the report is genuine, and we have included all out data/code/designs.
- We have not submitted any part of this work for another assessment.
- Our work did not involve human participants, their cells or data, or animals.

Contents

List of Figures

List of Tables

List of Algorithms

List of Symbols

s	sign
e	exponent
r, k	regime value
f	fraction
ES	exponent size
RS	regime size
P	posit value
EE	Exact Exponent
EED	Exact Exponent Difference
NaR	Not-a-Real number
FPU	Floating Point Unit
PPU	Posit Processing Unit
RTL	Register Transfer level

Acknowledgements

Thanks to Jeremy and William from Embecosm and our supervisor Graeme Bragg for their advice and support throughout.

Chapter 1: Introduction

1.1 Posits

The Posit standard, developed by Dr. John L. Gustafson, emerged as a compelling alternative to the prevalent IEEE-754 standard, originating from his arithmetic framework known as Universal Number (Unum) [2]. The design is geared towards achieving a broader dynamic range, improved accuracy, more efficient hardware due to its similarity to the two's complement integer format, along with simplified handling of exceptions. A key feature in the Posit format is the incorporation of the 'regime' bit, which is pivotal in dynamically allocating bits, thereby allowing for a more adaptable and efficient way to encode the scaling of a number. Moreover, it employs the concept of 'tapered' precision, which offers increased precision for numbers close to -1 and 1, while gradually decreasing it for extremely large or small numbers. This characteristic is particularly beneficial in various real-world computational applications, especially in machine learning domains such as Deep Neural Networks (DNNs), which typically utilise normalised parameters known as weights. The Posit format's enhanced precision near 1 allows for a more efficient bit usage compared to conventional floating-point formats, thereby enabling the use of smaller sizes. When operating at reduced bit widths, Posits have shown to provide improved accuracy and reduced power consumption relative to the usual 16-bit or 32-bit floating-point operations [3][4]. This aspect of Posits proves exceptionally advantageous in environments with limited resources, as the optimal utilisation of each bit can result in notable performance improvements.

1.2 Collaboration

The Group Design Project (GDP) involves collaboration with an external partner, Embecosm. Founded in 2008, this company primarily concentrates on four key areas: compiler toolchain development and porting, hardware modelling, open-source tool support and embedded operating systems. Additionally, Embecosm has expertise in specialised areas such as machine learning optimisation, super-optimisation (advanced compiler optimizations), energy efficiency in computing, and compilation for security [5]. Their overarching goal is to deliver top-tier compiler technology on a global scale. As Embecosm was a member of the OpenHW Group, an organisation that takes contributions from its members and independent engineers to provide a platform for open-source hardware development particularly in the RISC-V area [6], we would use their tools and be encouraged to interact with the open-source community.

1.3 Group Design Project Goal

In this project, our client, Embecosm, requested the development of a modified Floating Point Unit (FPU) that utilises the Posit standard. This revamped unit, which was named the Posit Processing Unit (PPU), was intended to replace the existing IEEE-754 based FPU in the given open-source RISC-V (cv32e40p) core. The primary aim was to modify or replace the existing RTL to create the PPU prototype using SystemVerilog and evaluate its potential advantages provided by the new representation of reals. Subsequent to its development, the PPU would undergo a comprehensive verification process, aligning with the Open Hardware Group's verification methodologies, followed by its integration into the cv32e40p. Additionally, the project plan involved the creation of a Verilator model for the entire System on Chip (SoC) and a stretch goal was set to implement the new core onto a Nexys-A7 FPGA [7] and conduct a series of tests to evaluate the effects of implementing a Posit-based FPU on the core and measure its performance.

The project was initially envisioned to unfold in several phases:

- Phase 1: Transitioning the instruction algorithms to Posit-based format
- Phase 2: Conducting verification using testbenches
- Phase 3: Integrating the PPU into the cv32e40p
- Phase 4: Adapting to the Open Hardware Group Verification Strategy

Some aspects of these initial plans were modified or scaled down due to project difficulty and the constraints in time and resources. Emphasis was placed predominantly on the project's critical components, particularly ensuring and verifying the correct functionality of the completed work through testbenches.

Chapter 2: Background

2.1 IEEE-754

Floating-point representation is essential for handling real numbers in computing, especially since these numbers cannot be perfectly represented in hardware due to their inherent infinite nature. This limitation necessitates rounding real numbers to a fixed number of digits, giving rise to the term ”floating-point”, which refers to the ability of the decimal point to shift to any position required for the number's representation. This method approximates real numbers within a specified range and precision, allowing for the expression of numbers that cannot be accurately captured as integers.

Among various standards, IEEE-754, established in 1985 and revised in 2008 [8], is the most widely adopted for real number operations and arithmetic. It provides a comprehensive framework, dictating the format for floating-point numbers, and defining how bits are allocated for sign, exponent and mantissa (or significand). In addition to standard number representation, it includes subnormal values for representing extremely small numbers. It also defines special values such as positive and negative zeros, infinities, and two types of NaN (Not a Number). Specifically, in a 32-bit configuration, it designates 1 bit for the sign, 8 bits for the exponent, and 23 bits for the mantissa. Consequently, the largest positive number is approximately 3.40×10^{38}, and the smallest positive subnormal positive number is about 1.40×10^{-45}. However, the standard does not include a mechanism for gradual overflow, resulting in the potential of large numbers abruptly overflowing to infinity [9]. Moreover, IEEE-754 specifies several rounding methods and the choice of rounding method can affect precision, particularly in the context of operations close to underflow [10] or overflow thresholds. The standard's rounding rules aim to minimise errors introduced during these operations, but some loss of precision is inevitable in edge cases.

2.2 Posit Standard

Posits and IEEE-754 Floating Points have common properties in their bit patterns, including a sign bit, a fractional component, and parts designated for the exponent. However, key differences are evident in their structures and interpretations. Notably, in the case of Posits, negative values are represented in two's complement manner, akin to integer binary numbers. This means that, aside from the sign bit, all other bits are inverted and plus one when representing negative numbers, which differs from the method used in IEEE-754 floating point representation. Figure 2.1 depicts the general binary representation of the Posits.

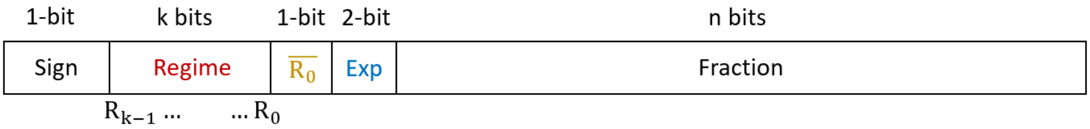


Figure 2.1: General Posit Binary Representation

In the Posit format, the exponent component is divided into two distinct parts: the regime and the exponent. The length of the exponent is governed by a parameter known as the Exponent Size (ES), which needs to be determined before conducting any Posit operations. According to the most recent Posit standard [11], the ES is generally set to 2, thereby allowing a maximum of a 2-bit exponent. This setting is optimised for general use but can be modified to suit specific needs. Opting for a larger ES value results in a wider dynamic range, while a smaller ES improves precision.

The calculation of a Posit value, denoted as P, from its bit pattern can be achieved using both equations 2.1 and 2.2(ES=2). These equations take into account the various components of a Posit number: the sign bit (s), the exponent size (ES), the regime (r), the exponent (e), and the fraction part (f).

$$P = (-1)^s \times (2^{2^{ES}})^r \times 2^e \times (1 + f) \tag{2.1}$$

$$P = ((1 - 3s) + f) \times 2^{(1-2s)\times(4r+e+s)} \tag{2.2}$$

The processes for managing the exponent and fraction in Posits resemble those used for the IEEE floating point system. For the exponent in Posits, its binary value is taken directly, eliminating the requirement for a bias as seen in the IEEE format. The fractional part, similarly, is represented as 2^{-n}, aligning with the IEEE floating point standard.

The core of the Posit standard is encapsulated in its regime, which essentially acts as a dynamic scaling factor for the exponent. This unique feature allows for flexible bit allocation within the Posits' bit representation, adapting to the magnitude of the number being encoded. The $2^{2^{ES}}$ in equation 2.1 is also referred to as the "useed".

The regime bits follow the sign bit in a Posit number, forming a continuous sequence of identical bits, all being either 1s or 0s, and ending with a terminating bit that differs from the rest of the regime. The actual value of the regime, which significantly influences the Posits' scale, is determined by the count of these consecutive identical bits, denoted as k. A positive regime is identified by a series of consecutive 1s or a terminating bit of 0. Its value is derived by subtracting one from the total count of these 1s. On the other hand, a negative regime is signified by a string of consecutive 0s or a terminating bit of 1. The value of a negative regime is computed as the negative count of these 0s. This relationship between the consecutive bit, R_0 and the corresponding regime value is captured in equation 2.3, providing a formula representation of how regime values are calculated.

$$r = \begin{cases} -k, \; R_0 = 0 \\ k-1, \; R_0 = 1 \end{cases} \tag{2.3}$$

Example 1: Consider a 32-bit Posit number: 0|11110|10|011011001110100101111000. The components are:

- Sign = 0
- Regime bits = 11110
- Exponent = 10
- Fraction = 011011001110100101111000

Since the regime's consecutive bits are 1s and their count is 4, the regime value is calculated as $4 - 1 = 3$. The exponent in binary is 10, which is 2 in decimal. For the fraction, an implicit hidden bit with the value of 1 is taken into account, making the fraction approximately 1.4254302. Applying equation 2.1, the Posit number's decimal value is approximately 23354.24.

Example 2: Consider a 32-bit Posit number: 1|11111110|10|010101111101010000111. Taking note that sign bit is 1, two's complement is performed. The components are:

- Sign = 1
- Regime bits = 00000001
- Exponent = 01
- Fraction = 101010000010101111001

With the regime's consecutive bits being 0s and their count being 7, the regime value is -7. The exponent is 1 in decimal. Considering the hidden bit 1, the fraction is approximately 1.6569213. By applying equation 2.1, the decimal value of the Posit number is around -1.2345×10^{-8}.

Some regime bits and their corresponding values are also tabulated in Table 2.1 for clarification.

Bits	0000	0001	001x	01xx	10xx	110x	1110	1111
Value	-4	-3	-2	-1	0	1	2	3

Table 2.1: Regime Bits with Corresponding Values

It is important to note that a regime value of 0 is represented by the bit sequence 10, implying that at least two bits are required for the regime in any Posit number. When the regime bits do not consume a significant portion of the total bit allocation, there is sufficient space for the exponent and fraction bits within the given bit size of the Posit number. However, special cases arise due to the dynamic allocation of regime bits.

For instance, in the context of a 32-bit Posit number, the regime bits could extend up to 31 bits, accounting for all but the sign bit. In such extreme scenarios, where the regime bits consume nearly the entire space, there would be no remaining room for the exponent and fraction. Consequently, these components are assumed to be zero. This results in a Posit number that represents a large magnitude, determined solely by the extensive regime as illustrated in Figure 2.2.

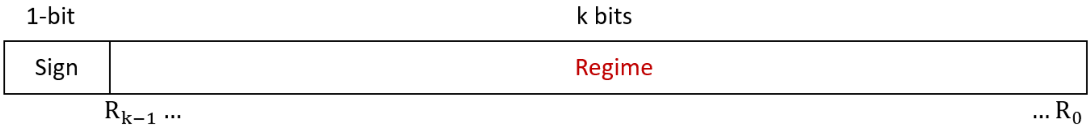


FIGURE 2.2: Large Regime, Fraction and Exponent Truncated

Conversely, in situations where available space after the regime is just enough to accommodate the exponent, the fraction part is set to 0. This leads to a number where precision is limited by the available bit space. Furthermore, in the case of the exponent, if the Posit number only allows for a single bit due to space constraints, a zero is appended to meet the 2-bit requirement. For example, an exponent pattern of 1x (where 'x' is an indeterminate bit) would default to 10 if there is not enough space as illustrated in Figure 2.3

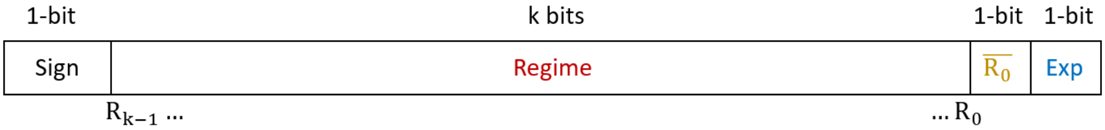


FIGURE 2.3: Large Regime with 1-bit Exponent, LSB of Exponent and Fraction Truncated

2.2.1 Advantages of Posit

Broader Dynamic Range

The range of numbers that Posit numbers can represent is influenced by the Exponent Size (ES). In Posit notation, the highest possible positive value is expressed as $maxpos = useed^{n-2}$, while the smallest positive value is denoted as $minpox = useed^{2-n} = \frac{1}{maxpos}$, where n refers to the total number of bits. It is important to note that a larger ES reduces the precision of the representation. For instance, in a 32-bit IEEE-754 format, the maximum positive value is roughly 3.4×10^{38}, and the smallest positive subnormal value is about 1.4×10^{-45}. In contrast, a 32-bit Posit with an ES of 2 can represent a maximum positive number of approximately 1.3×10^{36} and a minimum positive value of around 7.5×10^{-37}. This range appears narrower compared to the IEEE format.

However, increasing the ES by just 1 in the Posit format results in a significantly larger range. The maximum positive number then becomes around 1.7×10^{72}, and the smallest positive number becomes approximately 5.6×10^{-73}, albeit with a slight reduction in precision by one bit.

Tapered Precision

A key advantage of Posit is its tapered precision, where precision is highest near the value 1 and decreases with increasing or decreasing magnitude. This attribute is especially beneficial for many real-world calculations that commonly involve numbers close to 1. For instance, a 32-bit Posit with ES=2 can utilise up to a 27-bit fraction ($\approx$ 8.13 decimal digits of precision) as shown in Figure 2.4, compared to the 23-bit fraction ($\approx$ 6.92 decimal digits of precision) in IEEE floating points in Figure 2.5. This means that by normalising numbers within the range of 2^{-20} to 2^{16} (approximating $9.5367431640625 \times 10^{-7}$ to 65536 in decimal), programmers can achieve higher precision.

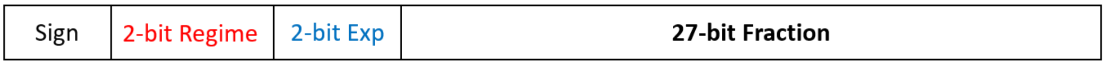


Figure 2.4: 32-bit Posits with 2-bit Regime and 27-bit Fraction

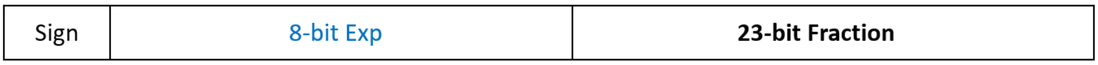


Figure 2.5: 32-bit IEEE-754 Floats with 23-bit Fraction

Interestingly, Posits redefine the concepts of ”overflow” and ”underflow”. Instead of overflowing to infinity, an increase beyond the maximum representable value in Posits leads to saturation, where the value ceases to increase. Similarly, for very small values, Posits gradually lose precision, approaching zero without abrupt underflow.

Reduced Rounding Errors

Due to the tapered precision and dynamic bit allocation, Posits tend to exhibit fewer rounding errors compared to traditional floating-point numbers. This results in more accurate computations, particularly in scenarios requiring high numerical precision.

Simpler Hardware Implementation

Posits streamline the management of special cases by including only zero and Not-a-Real (NaR), simplifying the necessary control logic for exception handling. Zero is uniquely represented by a sequence of all zeros, while NaR is signified by a leading 1 followed by zeros. In Posit arithmetic, any operation involving NaR invariably results in NaR. This includes scenarios traditionally associated with infinity or undefined results in other systems, such as division by zero or the square root of negative numbers. NaR effectively takes on the role of infinity and propagates through computations, indicating erroneous or undefined results without the need for signaling exceptions. This approach ensures the correctness of results can be directly inferred from their values. Furthermore, Posits do not include subnormal numbers, eliminating complex handling mechanisms for such cases. This absence contributes to more streamlined hardware designs, potentially enhancing computational efficiency while reducing power consumption.

Potential for Specialised Application

In fields such as scientific computing or machine learning, where precision around the value of 1 is often critical, Posit's tapered precision can be a significant advantage. This characteristic enables the use of smaller bit sizes while still maintaining high precision, offering a viable alternative to larger bit-sized floating-point numbers. Such an approach can result in more streamlined and effective hardware designs.

2.2.2 Potential Trade-offs

Precision Loss in Extreme Values

While Posits offer tapered precision that is beneficial for numbers close to 1, this feature can lead to precision loss for values at the extreme ends of the spectrum. As numbers become significantly large or small, the number of bits allocated to the fraction decreases, resulting in less precision. This aspect might limit the use of Posits in applications where high precision is required for very large or very small numbers but this can be mitigated by normalising these numbers in software.

Requirement for an Extraction Scheme

The dynamic nature of Posit numbers, specifically the variable-size regime, necessitates a mechanism to determine each component (regime, exponent, and fraction) at runtime. This adds complexity to the implementation, as it requires an extraction scheme to correctly parse the Posit number into its constituent parts. Implementing this extraction efficiently is crucial for performance, especially in hardware.

2.3 SoftPosit

SoftPosit [12] is a software-based implementation of Posit format, developed by the NGA team, the official group responsible for Posit development. It serves as a computational platform for researchers to investigate and explore the unique characteristics of Posits. It is particularly useful for simulating and examining Posit arithmetic within a software framework. It encompasses a wide range of functions dedicated to Posit arithmetic. These functions include, but not limited to, conversions between double-precision floating-point numbers and 32-bit Posits, as well as basic arithmetic operations like addition, subtraction, multiplication and division. Additionally, more complex operations such as fused multiply-add are also supported. These functionalities are not just practical for computational tasks but also serve as a theoretical basis for understanding and exploring the Posit format. In this project, the SoftPosit library played a crucial role in creating test sequences and validating the arithmetic operations as they were implemented and developed.

2.4 RISC-V

RISC-V is an open-source instruction set architecture developed by the University of California, Berkeley. As an instruction set, it follows Reduced Instruction Set Computer (RISC) principles, this means that the complexity of instructions are minimised as well as their size whilst simplifying the circuitry which is aimed to increase the performance of a RISC-V based processor. One of the many advantageous of RISC-V is it's flexibility, including many standard extensions to the instruction set and reserved instruction encodings for full custom instructions. Currently the standard includes 32-bit and 64-bit address spaces with a potential future 128-bit standard [1].

Turning ten years old in 2024, RISC-V is becoming more widely adopted with many large companies in the field developing RISC-V based hardware, this combined with its open-source nature makes it extremely useful to carry out fully open-source projects to experiment with using the instruction set.

2.5 The cv32e40p

The cv32e40p is part of OpenHW's CORE-V family of open-source RISC-V cores [13], it is an integral part of the PULP (Parallel Ultra Low Power) platform, an open-source initiative focused on developing high-performance and energy-efficient computing systems. This 32-bit RISC-V core is designed with an in-order 4-stage pipeline architecture, encompassing Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), and Write-Back (WB) stages. It supports the RV32I Base Integer Instruction Set and includes several standard extensions, crucial among which is the single-precision Floating-Point (”F”) extension using floating-point (F) registers. This extension is essential for operations that require IEEE-754 standard floating-point arithmetic. Most of the instructions from the ”F” extension are implemented in the co-processor (cvfpu) including with the core with the exception of the load, store and move instructions.

A key feature of the cv32e40p is its modular design, allowing for the Floating Point Unit (FPU) to be activated through the “FPU” parameter in the cv32e40p_top top-level module. This modularity facilitates the integration of our project's components. To incorporate the Posit Processing Unit (PPU) into the cv32e40p, the PPU modules must be included in the manifest list and the PPU top level module must be instantiated in a wrapper module for the cv32e40p_top. This process effectively replaces the conventional IEEE-754 based FPU with our newly implemented PPU modules, aligning with the project's goal to explore and harness the potential of Posit arithmetic within the framework of the cv32e40p core.

2.6 Literature review

Over time, there has been a significant increase in research focused on Floating Point and Posit representations. This has resulted in numerous studies examining different aspects of these numerical systems. Investigations have spanned from the verification of fundamental instructions in both FP and Posit units to specific studies on the Posit Enabled RISC-V Core (PERI). Additionally, some research has concentrated on integrating FP and Posit directly with the hardware components of a chip. This section offers an overall review of the academic literature.

A vast array of academic papers have been dedicated to the design and verification of algorithms and arithmetic processes, utilising three primary methods for this purpose: test vector generation, Softposit verification [12], and a methodology developed by [14]. K.Nouh and colleagues [15] conducted verification of binary floating-point operations, including Addition, Subtraction, and Multiplication, using randomly generated testing vectors. These vectors were produced by resolving predefined constraints with tools like SystemVerilog and QuestaSim. Such a predefined constraint-based approach is key in examining edge cases in floating-point operations. A similar methodology was employed by A.S.Ahmed et al. [16] in their verification of floating point square-root algorithms. However, this technique did not entirely encompass corner case scenarios, leading to partial verification in some instances.

Conversely, the Softposit-based verification approach has been adopted in various research works. Notably, H.Zhang and team [17] applied Softposit along with a comprehensive range of vectors to validate their posit multiply-accumulate unit generator. Similarly, Sugandha Tiwari and colleagues [18] used Softposit to generate random testing inputs for the verification of their Posit unit. They paid particular attention to corner cases like Zero and NaR, rigorously testing these scenarios using the Softposit verification methodology.

Additionally, several studies have implemented the verification technique, Julia introduced by [14]. M. K. Jaiswal and his team, in their works [19] and [20], employed this strategy to verify the Posit unit and the Posit addition/subtraction blocks, respectively. This method was also used in the verification of Posit addition and multiplication instructions within "Parameterized Posit Arithmetic" units, as documented in [21]. Furthermore, the study cited as [22] specifically focuses on the conversion process between IEEE-754 floating point and Posit formats, highlighting the versatility and application range of these methodologies.

In the domain of hardware-level research, several studies have ventured into the integration of either Floating Point Units (FPU) or Posit Processing Units (PPU) onto Field Programmable Gate Array (FPGA) devices. Following the successful testing of the Posit Arithmetic Core Generator (PACoGen), M. K. Jaiswal and his team proceeded to implement this generator on a Xilinx Virtex-7 FPGA, where the integrated hardware system demonstrated complete functionality. Similarly, after employing Softposit in their verification stage, Sugandha Tiwari and colleagues ([18]) synthesized the PERI onto an Artix-7 FPGA device, resulting in a fully operational core that functioned accurately at a frequency of 100 MHz.

In short, the development of the Posit standard and the exploration of its advantages remain a work in progress. Therefore, this project seeks to help contribute to this ongoing effort by creating a Posit Processing Unit (PPU) capable of performing a range of arithmetic operations, including addition, subtraction, multiplication, division, and square root. The verification of this PPU will employ the Softposit method along with Questasim UVM. Subsequently, the PPU will be integrated with the cv32e40p to conduct comprehensive tests. This project not only advances our understanding and application of the Posit standard but also lays the groundwork for future enhancements of the PPU's capabilities and its potential integration into FPGA systems for more in-depth analysis and application.

Chapter 3: Methodology

3.1 Redesigning the FPU

To adapt the floating point unit (FPU) on the cv32e40p [23] to the posit processing unit (PPU), it is necessary to revise the underlying algorithms for the FPU instructions, with reference to the RISC-V ”F” standard extension [1] for single-precision floating point operations as they are implementeed in the FPU. The specifications for the posit standard are outlined in [11]. This adaptation will encompass arithmetic operations like addition, subtraction, multiplication, division, and square root, all formatted in posit, along with other tasks such as classification, sign injection, comparison, and conversion as detailed in Table 3.1.

Operation Group	**Instruction**	**Description**
NONCOMP	SGNJ.S	Sign injection - Copy sign
	SGNJX.S	Sign injection - XOR signs
	SGNJN.S	Sign injection - Copy inverted sign
	FMIN.S	Minimum
	FMAX.S	Maximum
	FEQ.S	Equal Comparison
	FLT.S	Less-than Comparison
	FLE.S	Less-than or Equal Comparison
	FCLASS.S	Classification
ADDMUL	FADD.S	Addition
	FSUB.S	Subtraction
	FMUL.S	Multiplication
	FMADD.S	Multiply-Add
	FMSUB.S	Multiply-Subtract
	FNMADD.S	Negative Multiply-Add
	FNMSUB.S	Negative Multiply-Subtract
DIVSQRT	FDIV.S	Division
	FSQRT.S	Square Root
CONV	FCVT.W.S	Posit to signed integer
	FCVT.S.W	Signed integer to posit

Table 3.1: RISC-V ”F” Standard Extension Instructions

Within the existing architecture of the cv32e40p's FPU [24], these operations are structured into four primary blocks: the addition-multiply (ADDMUL) block, the division-square root (DIVSQRT) block, the non-computational (NONCOMP) block, and the conversion (CONV) block. Accordingly, the PPU will also adopt this organisational framework, as illustrated in Figure 3.1.

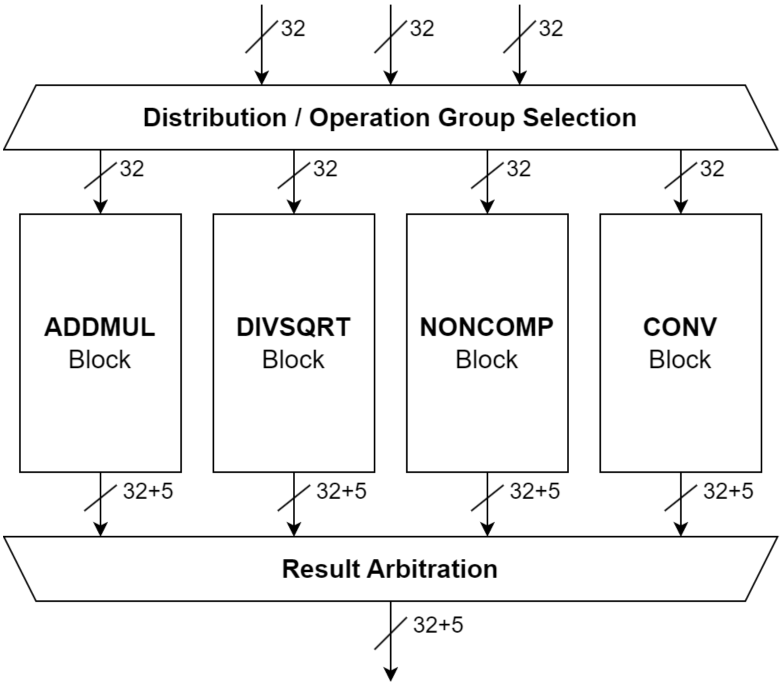


FIGURE 3.1: Components of the Posit Processing Unit

The ADDMUL block is distinguished by its requirement for three operands, in contrast to other groups that only require two. Along with the result, it returns a 5-bit status indicator that is consistent with RISC-V's existing framework. This status includes indicators for various conditions: [NV, DZ, OF, UF, NX], representing invalid operation, division by zero, overflow, underflow, and inexact result respectively. In the context of Posit arithmetic, overflow (OF) and underflow (UF) are not applicable, and the invalid (NV) and inexact (NX) indicators are not utilised. The division by zero (DZ) flag signals a division by zero, with the outcome being set to NaR (Not a Real number). To streamline development, each operation unit's algorithm was first developed separately from the core-v architecture. This approach allowed for thorough validation through testbenches before their integration into the core system.

3.1.1 Posit Extraction

Extraction is the first step for Posit computational operations. As described from lines 1 to 10 of Algorithm 1, the first thing that needs to be done is to check exceptions, ∞/NaR or 0. The bit pattern for ∞ is 100...000 and for zero, it has the bit pattern 000...000. Considering this, the second MSB to LSB are checked to see whether all of them are '0'. After checking the MSB, the corresponding flag (zero_Check) is raised if either of them is detected. As introduced in section 2.2, Posit is divided into sign, regime, exponent and fraction. The bits immediately after the sign bit are registered in 'InRemain' and negated (two's complement) if the sign bit is 1, indicating the posit input is negative. The regime bits consist of a sequence of identical consecutive bits followed by one bit that is the opposite. Subsequently, the InRemain register is left shifted to flush the regime bits. If there are still bits left, two bits are first allocated to the exponent and the rest are assigned to the fraction. If there are not enough bits, zeros are filled into registers of exponent and fraction bits.

Algorithm 1 Posit Extraction

```
Input: 32-bit Posits
Output: Sign, k, Exponent, Fraction, InRemain, inf, zero
Exception check
if Second MSB to LSB are all 0 then
    zero_Check ← 1
end if
if MSB is 1 and zero_check is 0 then
    inf ← 1
end if
if MSB is 1 and zero_check is 0 then
    zero ← 1
end if
Extraction
if MSB of the input is 1 then                        ▷ negative input
    InRemain ← -InRemain[N-2:0]
else
    InRemain ← InRemain[N-2:0]
end if
if RegimeCheck is 1 then                             ▷ regime bit pattern of 111...110
    k ← EndPosition - 1                              ▷ equation 2.3
else                                                 ▷ regime bit pattern of 000...001
    k ← -EndPosition
end if
ShiftedRemain ← InRemain << (EndPosition + 1)        ▷ flush regime bits
Exponent ← Most significant 2 bits of ShiftedRemain
Fraction ← concatenation of 1-bit hidden one, remaining bits from ShiftedRemain,
    filling zeros to fill the declared bit length
```

The 'regimeCheck' flag plays a critical role in identifying whether the regime bits consist of consecutive ones or zeros, and 'k' represents the regime value. When 'regimeCheck' equals 1, 'k' is calculated as the count of consecutive bits minus one. Conversely, if 'regimeCheck' is 0, 'k' becomes the negative value of the count of consecutive bits. Essentially, this module processes a 32-bit posit by breaking it down into its sign, regime value (k), and fraction components. It is noteworthy that the 'k' and 'InRemain' variables are of signed type, and the fraction is concatenated with a 1-bit hidden bit, Additionally, it directly checks for special values such as zero and infinity/NaR, subsequently outputting the appropriate flag bits.

3.1.2 Addition and Multiplication

The Addition and multiplication instructions are implemented in the ADDMUL block. This includes the instructions FADD.S, FSUB.S, FMUL.S, FMADD.S, FMSUB.S, FNMADD.S and FNMSUB.S.

3.1.2.1 Addition/Subtraction Arithmetic

As outlined in Algorithm 2, prior to executing the arithmetic computation, it is necessary to perform checks for exceptions and validate the exact operation to be performed. This validation depends on the sign of each input and the specific instruction being executed, as detailed in Table 3.2.

Algorithm 2 Addition/Subtraction Arithmetic

```
Input: Sign, k, Exponent, Fraction, InRemain, inf, zero from both inputs
Output: Added/Subtracted fraction, normalized Added/Subtracted fraction, exact exponent, LS, inf flag, zero flag
 1: Exception Check:
 2: inf ← inf1 | inf2
 3: zero ← zero1 | zero2
 4: Add/Sub:
 5: op ← $\overline{Sign1 \oplus (instruction \oplus Sign2)}$
 6: Greater_Than ← (InRemain1[N-2:0] > InRemain2[N-2:0]) ? 1 : 0
 7: LS, LR, LE, LM ← sign, k, Exponent, fraction from Larger input
 8: SS, SR, SE, SM ← sign, k, Exponent, fraction from Smaller input
 9: R_diff ← LR - SR
10: E_diff ← LE - SE
11: EDD ← R_diff × ($2^{ES}$) + (E_diff)                          ▷ Equation 3.1
12: if EDD > 2N then
13:     SM_sft_temp ← extended SM >> 2N
14: else
15:     SM_sft_temp ← extended SM >> EDD
16: end if
17: SM_sft ← SM_sft_temp[first N-1 bit] and other bits are OR together as LSB
18: if op then                                                    ▷ Table 3.2
19:     Add_Mant ← LM1 + SM_sft
20: else
21:     Add_Mant ← LM1 − SM_sft
22: end if
23: Normalization:
24: LBD_in ← concatenate (Add_Mant[2*N] | Add_Mant[2N-1]) and Add_Mant[2N-2:N]
25: if fraction overflows then
26:     Add_Mant_sft ← Add_Mant[2N:1] << shift
27: else
28:     Add_Mant_sft ← Add_Mant[2N-1:0] << shift
29: end if
30: Add_Mant_N ← Add_Mant_sft
```

Table 3.2 presents a systematic approach to determine the correct arithmetic operation based on the sign bits of two inputs and the given instruction, which is executed in line

5. When both signs are “0” (indicating both inputs are positive), an “Add” instruction leads to the standard operation of addition and a “Sub” instruction results in subtraction. However, if the sign bit of the first input is “0” (positive) and the second input is “1” (negative), an “Add” instruction actually subtracts the second input from the first, reflecting the subtraction of a negative number. Similarly, a “Sub” instruction in this scenario adds the second input to the first, as subtracting a negative is equivalent to adding a positive. The opposite is true when the first input’s sign bit is “1” and the second’s is “0”: “Add” performs a subtraction, and “Sub” performs an addition, again due to the inversion introduced by the negative sign. When both inputs are negative, indicated by their sign bits being “1”, the “Add” instruction combines the magnitudes of the two negatives, which is still and addition, and “Sub” separates them, which is a subtraction. This Table ensures that the arithmetic logic accurately interprets and processes the sign bit to execute the correct mathematical operation as intended by the instruction.

Sign of input 1	**Sign of input 2**	**Instruction**	**Operation**
0	0	Add	Addition
0	0	Sub	Subtraction
0	1	Add	Subtraction
0	1	Sub	Addition
1	0	Add	Subtraction
1	0	Sub	Subtraction
1	1	Add	Addition
1	1	Sub	Subtraction

Table 3.2: Operation Based on Inputs’ Sign and Instruction

Afterwards, as shown in lines 6 to 10 of Algorithm 2, the two inputs are compared to ascertain which is greater. Following this comparison, the fraction bits of the smaller input are left shifted to properly align the fraction parts for both inputs, facilitating the subsequent addition or subtraction. This shift is based on the exact difference in exponents between the two inputs, referred to as exact exponent difference (EED). The EED in the Posit format incorporates both regime and exponent differences, which are denoted as ’R_diff’ and ’E_diff’, respectively. This relationship is conveyed in equation 3.1:

$$EED = \log_2((2^{2^{ES}})^{R\_diff} \times 2^{E\_diff}) = r\_diff \times 2^{ES} + e\_diff \tag{3.1}$$

Specifically, for an exponent size (ES) of 2, the EED simplifies to:

$$EDD = 4 \times r\_diff + e\_diff$$

As an example, consider two Posit numbers (magnitude) represented as follows:

$$P_1 = 2^{4r_1+e_1} \times (1.f_1)$$

$$P_2 = 2^{4r_2+e_2} \times (1.f_2)$$

Let's assume:
For $P_1 : r_1 = 5, e_1 = 2, F_1 = 1.f_1$
For $P_2 : r_2 = 3, e_2 = 1, F_2 = 1.f_2$
The EDD is calculated as: $EDD = 4 \times 2 + 1 = 9$. Consequently, the addition of P_1 and P_2 is expressed as:

$$P_{\text{add}} = 2^{4r_1+e_1} \times \left((1.f_1) + \frac{1.f_2}{2^{\text{EDD}}} \right), \text{ where EDD} = 9$$

It is important to note that in this case, the r_1, r_2 and f_1 are the greater values, as assigned in the initial steps (lines 6 to 8) of Algorithm 2.

Building on the procedures detailed in lines 12 to 17, it becomes evident that magnitude or the amount of shifting is constrained by the maximum length of the intermediate (temporary) registers used in the process. When EED exceeds the size of those registers, the shift amount is limited to their maximum length. The most significant N-1 bits are then directly taken as the most significant N-1 bits of the arithmetic result, while the remaining bits are aggregated using OR operation to determine the LSB. This approach is designed to safeguard the accuracy of rounding within the computation.

Following that, the shifted fraction bits from the smaller input are added or subtracted from the greater input and overflow is checked for the outcome. Furthermore, as shown in the example below, some most significant bits may be lost when doing subtraction between two close numbers. In this case, the computed outcome is left shifted until the MSB is a '1' for a uniform representation.

$$\begin{array}{cccccccccc}
 & & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\
 & - & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\
\hline
 & & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
<< shift & & & & & & & & & \\
\hline
 & & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0
\end{array}$$

After the fraction bits have been calculated, regime and exponent bits need to be deduced for composing a Posit bit pattern. As shown in equation 3.2, the exact exponent (LE_O) is calculated using the sum of the concatenated regime (LR) and exponent (LE) from the larger input, adjusted for any overflow from the fraction computation and shifts induced by the subtraction.

$$LE\_O = \{LR, LE\} + overflow - shift \tag{3.2}$$

Both the LE_O and LR are designated as signed integers. Specifically, LR is a 6-bit number, aligned with the regime size (RS), while LE is a 2-bit number, according to the exponent size (ES).

The term "overflow" in this context relates to the carry bit resulting from the fixed-point arithmetic operations. This is linked to the Posit format's structure, which incorporates a 1-bit implicit hidden bit in the fraction part. Consequently, the integer magnitude of operands in addition or subtraction operations are limited to ensure they do not exceed a value of 2. This restriction results in a maximum potential output below 4, leading to at most a single carry bit or "overflow".

For example, consider the addition of two fractions, each approaching a magnitude of 2. The resulting sum would approximate 3 (binary representation: 11) but would never reach 4. Since the fraction only permits a single hidden bit, this overflow is adjusted into the calculation of LE_O. The formula also compensates for shifts occurring during alignment phase, especially in subtraction operands involving close valued operands. This ensures the resultant fraction accurately reflects the computed values' magnitude and precision.

The effectiveness of this formula lies in the binary concatenation of LR and LE {LR, LE}. This concatenation essentially multiplies the regime value by 4 and then appends the 2-bit exponent, creating a unified exponent representation calculated as $4 \times regime + exponent$. For instance, with LR as 3 and LE as 2, the concatenation becomes 000011|10, equivalent to $4 \times 3 + 2 = 14$ in decimal.

This remains applicable for negative regimes. For example, if LR is -4 and LE is 3, the concatenation results is 111100|11, corresponding to -13, which aligns with the arithmetic of $4 \times -4 + 3$. Therefore, by accounting for potential overflow and subtracting any shift adjustments, the formula effectively determines the total exact exponent for Posit addition and subtraction.

Next, the corresponding regime and exponent are split from the LE_O as demonstrated in Algorithm 3, where R_O is the count of identical consecutive bits in regime and E_O is the exact bit pattern of exponent part.

Algorithm 3 outlines the process for handling the value of LE_O. Initially, the algorithm checks if LE_O is positive; if it is not, it is negated (two's complement) to form LE_ON, otherwise, it remains unchanged, as illustrated in lines 1 to 6. When LE_O is positive, extracting R_O and E_O is relatively straightforward. E_O is obtained by taking the least significant ES bits from LE_O, while R_O is derived from the most significant RS bits plus 1. The addition of 1 in the case of R_O is because it represents the count of consecutive identical bits; for a positive regime, the regime value is one less than R_O. Therefore, to find R_O from the regime, 1 is added (R_O = positive_regime + 1).

The process becomes more nuanced when LE_O is negative due to the implications of two's complement representation for negative numbers. The extraction of E_O remains the same as in the positive case, involving the retrieval of ES bits from the LSB of LE_O. For R_O, however, adjustments are needed if LE is non-zero (01, 10, or 11 in binary). In such cases, 1 must be added to the extracted RS-bit MSB of LE_O to offset the effect of the two's complement negation. This compensation accounts for the contribution of the LE bits to the negation process. For instance, with LR = -2 and LE = 01, LE_O would be 111110|01, and LE_ON would then be 000001_11. Extracting the RS-bit MSB gives 000001, and with LE being non-zero, adding 1 results in 000010 (2 in decimal).

Algorithm 3 Post Addition Composition

```
Input: Regime from greater input, Exponent from greater input, Fraction overflow, the
       amount of left shift
 1: Sign Check:
 2: if LE_O is negative then
 3:     LE_ON ← -LE_O
 4: else
 5:     LE_ON ← LE_O
 6: end if
 7: Exponent:
 8: if LE_O is negative and last ES bits in LE_ON are not all 0 then
 9:     E_O ← Least significant ES bits of -LE_ON
10: else
11:     E_O ← Least significant ES bits of LE_ON
12: end if
13: Regime:
14: if LE_O is positive then
15:     R_O ← (Most Significant RS bits of LE_ON) +1
16: else if LE_O is negative and last ES bits in LE_ON are not all 0 then
17:     R_O ← (Most Significant RS bits of LE_ON) +1          ▷ L9 Compensation
18: else
19:     R_O ← Most Significant RS bits of LE_ON
20: end if
```

Conversely, if LE is 00, such compensation is unnecessary. For example, if LR = -2 and LE = 00, LE_O would be 111110|00, making LE_ON 000010|00. Extracting the RS-bit MSB yields 000010 (2 in decimal) directly, with no need for further adjustment, as two's complement inherently accounts for this due to LE being all zeros. Essentially, for negative LE_O or LR, R_O corresponds to the magnitude of the original negative LR value. The relationship between the LE and compensation is tabulated in Table 3.3.

Exponent Bits	**Two's compliment**	**Compensation**
00	(1)00	0
01	11	1
10	10	1
11	01	1

Table 3.3: Compensation for Negative Regime Depending on Exponent

3.1.2.2 Multiplication Arithmetic

In Posit arithmetic multiplication, equation 3.3 outlines the general computation of the magnitude, particularly noting that the Exponent Size (ES) is fixed at 2.

$$P_1 = 2^{4r_1} \times 2^{e_1} \times (1.f_1)$$

$$P_2 = 2^{4r_2} \times 2^{e_2} \times (1.f_2)$$

$$P_1 \times P_2 = 2^{4(r_1+r_2)} \times 2^{(e_1+e_2)} \times (1.f_1 \times 1.f_2) \tag{3.3}$$

The results of both the regime and exponent are calculated by summing these respective components from two input numbers. Regarding the fraction part, this operation is performed as fixed-point arithmetic in SystemVerilog. A unique aspect of Posit numbers emerges during the multiplication of fraction bits: the existence of an implicit leading "1" in the fraction. This bit implies that the multiplication of the fraction part ranges from [1,4) in binary terms. Consequently, managing "overflow" during fractional multiplication becomes a crucial aspect of this process. It is important to note that this type of overflow does not imply exceeding the Posit format's maximum representation value. Rather, it refers to scenarios where the integer part of the product could reach 2 or more, but remain under 4, due to the implicit leading "1". As explained with the example below, the binary number 1100, representing 1.5 in decimal, and 1011, representing 1.375, combine to yield 2.0625, necessitating the use of the most significant bit (MSB) of the output, indicating an overflow. Considering this, the MSB of the outcome should be checked after two fraction parts are multiplied, and the Post Multiplication Composition is done as shown in Algorithm 4.

$$\begin{array}{cccccccc}
 & & & & 1 & 1 & 0 & 0 \\
\times & & & & 1 & 0 & 1 & 1 \\
\hline
1 & 0 & 0 & 0 & 0 & 1 & 0 & 0
\end{array}$$

Algorithm 4 addresses this overflow by checking the MSB of the fractions' product. Detected overflows are handled by incrementally modifying the exponent. If this adjustment leads to an exponent overflow, further modifications are made to the regime part. Such measures are vital to maintain the Posit number format's integrity during multiplication.

Algorithm 4 Post Multiplication Composition

```
Input: Regime from both inputs, Exponent from both inputs, MSB from multiplied
    outcome
sumE = e1 + e2 + MSB of the multiplied outcome
sumR = r1 + r2 + sumE_Ovf
E_O ← least significant ES bits of sumE
if sumR is negative then
    R_O ← -sumR
else
    R_O ← sumR +1
end if
```

3.1.2.3 FMA Arithmetic

Table 3.4 presents how the "op_mod" control bit from the CPU modifies the standard Fused Multiply-Add (FMA) operation, which combines multiplication in section 3.1.2.1 and addition in section 3.1.2.2 in a single step to reduce rounding errors.

Instruction	op_mod	Operation	Adjustment
FMADD	0	FMADD	none
FMADD	1	FMSUB	Invert sign of operand C
FNMSUB	0	FNMSIB	Invert sign of operand A
FNMSUB	1	FNMADD	Invert sign of operands A and C
ADD	0	ADD	Set operand A to 1
ADD	1	SUB	Set operand A to 1, Invert sign of operand C
MUL	0	MUL	Set operand C to 0

Table 3.4: Effect of op_mod Signal on FMA Operation

Normally, FMA uses three operands (A, B, and C) for the calculation P=A×B+C. Depending on the ”op_mod” value, the FMA instruction can perform different operations: a zero value means a standard FMA operation, while a value of one adjusts the operation by either inverting the sign of certain operands or changing the operation entirely—from addition to subtraction or vice versa, as depicted in Table 3.4. This allows the FMA module to carry out a range of operations: for instance, it can perform an FMADD or switch to FMSUB by inverting the sign of C, or it can change an FNMSUB to an FNMADD by inverting the signs of both A and C. Additionally, it can alter an ADD operation to a SUB by inverting the sign of C, or it can execute a simple multiplication by setting C to zero. This flexibility enables the CPU to dynamically choose the appropriate operation for a given computational task.

3.1.3 Division and Square Root

The two division and square root instructions, FDIV.S and FSQRT.S are implemented in the DIVSQRT block.

3.1.3.1 Division Arithmetic

Like the multiplication unit, division involving fraction bits is executed using fixed-point arithmetic. Due to the presence of an implicit hidden bit set to 1 in the fraction part, the results from any fractional division will fall within the [1,2) range. To facilitate the fixed-point arithmetic for division, the fraction bits of dividend undergo a left shift and are then stored in a 96-bit register, as detailed in Algorithm 5. The purpose of this left shift is to generate a 64-bit output representing the fraction part of the divisin result.

The process of composing the regime and exponent bits in division operations is analogous to that in multiplication, with the primary distinction being the replacement of addition operations with subtractions. This approach is elaborated in equation 3.4 and further detailed in Algorithm 6. The method employed to determine the output of R_O in division mirrors the technique used for handling LE_O in previous discussions, albeit with different specific terms.

Algorithm 5 Division Arithmetic

```
Sign ← Sign1 ˆ Sign2                                             ▷ Produce Output Sign
dividend ← fraction1 << 64                                       ▷ Shift dividend left 64 bits
diviser ← fraction2
Div_frac ← dividend / divisor                     ▷ SystemVerilog built-in divide function
Div_frac_temp_large ← Div_frac << 29
Div_frac_temp ← {Div_Frac_temp_large[95:31], |Div_frac_temp_large[32:0]}
if (MSB of Div_frac_temp) then               ▷ Normalize fraction bit and check underflow
    Div_frac_N ← Div_frac_temp
    Div_frac_underflow ← 0
else
    Div_frac_N ← Div_frac_temp << 1                                       ▷ Underflow case
    Div_frac_underflow ← 1
end if                                ▷ Div_frac_N is the normalized output fraction bit
```

$$P_1 \div P_2 = 2^{4(r_1-r_2)} \times 2^{(e_1-e_2)} \times (1.f_1 \div 1.f_2) \tag{3.4}$$

Algorithm 6 Post Division Composition

```
sumE ← e1 - e2 + Div_frac_underflow
sumR ← r1 - r2
Total_EO ← sumR << ES + sumE
if MSB of Total_EO then                                             ▷ Normalize Total_EO
    Total_EON ← -Total_EO
else
    Total_EON ← Total_EO
end if
if ( Total_EO[RS+ES+4]) then ▷ +ve Total_EO ▷ Extract regime from Total_EON
    R_O ← Total_EON[ES+RS+3:ES] + 1;
else if (Total_EO[RS+ES+4] & —(Total_EON[ES-1:0])) then
    R_O ← Total_EON[ES+RS+3:ES] + 1;
else
    R_O ← Total_EON[RS+ES+3:ES];
end if
```

3.1.3.2 Square-Root Arithmetic

The computation of a posit-format floating-point numeral is determined by equation 2.1. Given that square root calculations are valid solely for non-negative numbers without complex numbers implementation, and considering that the exponent's size is selected as 2-bit value for 32-bit posit number, S is set to 0 and ES to 2. This condenses the original formula to expression 3.5. Besides, it is crucial to recognise that if the operand is a special value, the outcome will vary accordingly: a zero operand yields a result of zero, while a sign bit of 1, indicating a negative value or Not-a-Real (NaR) number, will also result in a NaR outcome.

$$P = (16)^r \times (2)^e \times (1 + f) \tag{3.5}$$

Moreover, as the square root operation essentially applies a $\frac{1}{2}$ exponent to each segment of the equation, this modifies the formula to equation 3.6.

$$P = (16)^{\frac{r}{2}} \times (2)^{\frac{e}{2}} \times (1 + f)^{\frac{1}{2}} \tag{3.6}$$

Consequently, it is possible to independently deduce the square root for each individual part of the posit number. However, the equation 3.6 is only effective when both regime, r and exponent, e are even numbers. If not, they result in decimal values. Hence, an adjustment through rounding is necessary, and the decision here is to round downwards. Therefore, it is established that R represents the value derived from the downward rounding of r in cases where r is an odd number and e is even. Following the manipulation of exponents within the equation 3.6, it is observed that $\frac{r}{2}$ is effectively replaced by R, and $\frac{e}{2}$ is added with 2, as illustrated in equation 3.7.

$$\begin{aligned} P &= (16)^{R+\frac{1}{2}} \times (2)^{\frac{e}{2}} \times (1 + f)^{\frac{1}{2}} \\ P &= (2)^{4R} \times (2)^{2} \times (2)^{\frac{e}{2}} \times (1 + f)^{\frac{1}{2}} \\ P &= (16)^{R} \times (2)^{\frac{e}{2}+2} \times (1 + f)^{\frac{1}{2}} \end{aligned} \tag{3.7}$$

Subsequently, in scenarios where r is even and e is odd, E shall denote the value after rounding down e. Consequently, the regime remains at $\frac{r}{2}$, while $\frac{e}{2}$ is substituted by E, and an additional term of $\sqrt{2}$ is introduced, as depicted in equation 3.8.

$$\begin{aligned} P &= (16)^{\frac{r}{2}} \times (2)^{E+\frac{1}{2}} \times (1 + f)^{\frac{1}{2}} \\ P &= (16)^{\frac{r}{2}} \times (2)^{E} \times \sqrt{2} \times (1 + f)^{\frac{1}{2}} \end{aligned} \tag{3.8}$$

Lastly, when both r and e are odd, a similar approach of rounding down and manipulating exponents is employed, resulting in the inclusion of $\sqrt{2}$, as shown in equation 3.9.

$$P = (16)^{R} \times (2)^{E+2} \times \sqrt{2} \times (1 + f)^{\frac{1}{2}} \tag{3.9}$$

Table 3.5 presents an overview of the formulae for the regime, exponent, and fraction, (F=1+f) components of the posit square root across four distinct scenarios.

Once the square roots of the regime and exponent have been calculated, the main task is to compute the square root of the fractional part. There are various methods for this purpose, including the non-restoring square root algorithm and the Newton-Raphson algorithm.

Initially, the Newton-Raphson algorithm [25] is employed due to more intuitive formula to calculate the square root of fractions, denoted as $\sqrt{F}$. This iterative technique is

Regime, r	Exponent, e	Sqrt Regime	Sqrt Exponent	Sqrt Fraction
Even	Even	r/2	e/2	$\sqrt{F}$
Even	Odd	r/2	E	$\sqrt{2F}$
Odd	Even	R	e+2	$\sqrt{F}$
Odd	Odd	R	E+2	$\sqrt{2F}$

TABLE 3.5: Summary of Formulae for Each Component in Posit Square Root

known for its ability to rapidly produce increasingly accurate approximations of the square root, thanks to its fast quadratic convergence rate. This rate signifies that with each iteration, the number of accurate digits typically doubles. However, the effectiveness of the Newton-Raphson method is heavily reliant on the initial approximation of the square root. A closer starting point to the true root significantly reduces the number of iterations required for an accurate result. Moreover, a well-chosen initial approximation also diminishes the likelihood of divergence, a situation where the algorithm fails to locate the root, and prevents convergence to an incorrect root.

The Newton-Raphson method offers two approaches for implementation: direct calculation of the square root and calculation using the reciprocal of the square root [26]. The direct approach, as detailed in equation 3.10, involves dividing the fraction, F, for which the square root is sought, by an initial approximation of the square root, x. This method is simple and intuitive but may not be as effective when dealing with very small numbers, where division by a small approximation can cause substantial issues with precision.

$$x_{n+1} = \frac{1}{2}(x_n + \frac{F}{x_n}) \tag{3.10}$$
$$\lim_{n\to\infty} x_n = \sqrt{F}$$

In contrast, the reciprocal square root method, as per equation 3.11, opts for multiplication over division. It calculates the multiplication of the reciprocal square root approximation, $\frac{1}{\sqrt{x}}$ with F. This approach not only yields more consistent outcomes but also takes advantage of the fact that multiplication is typically quicker than division.

$$\frac{1}{\sqrt{x}_{n+1}} = \frac{1}{2}\frac{1}{\sqrt{x}}(3 - F(\frac{1}{\sqrt{x}})^2) \tag{3.11}$$
$$\lim_{n\to\infty} \frac{1}{\sqrt{x}_n} = \frac{1}{\sqrt{F}}$$

By determining the reciprocal of the square root and then multiplying it by the number, the desired square root could be derived efficiently: $\sqrt{F} = \frac{1}{\sqrt{F}} \times F$, making this method the preferred choice.

Moreover, it is noteworthy that an additional adjustment is required when the exponent e is odd, involving a multiplication by the $\sqrt{2}$ to modify the formula appropriately as depicted in equation 3.12.

$$\frac{\sqrt{2}}{\sqrt{x}}_{n+1} = \frac{1}{\sqrt{2x}}(3 - 2F(\frac{1}{\sqrt{2x}})^2) \tag{3.12}$$

Therefore, the two formulae can be generalised in equation 3.13.

$$\frac{1}{\sqrt{F}} = \lim_{n\to\infty} A\left(3 - B(C)^2\right) \text{ where } \begin{cases} A = \frac{1}{2\sqrt{x}}, B = F, C = \frac{1}{\sqrt{x}} & e \text{ is even} \\ A = \frac{1}{\sqrt{2x}}, B = 2F, C = \frac{1}{\sqrt{2x}} & e \text{ is odd} \end{cases} \tag{3.13}$$

Consequently, two separate lookup Tables have been created to cater to this need: one for even exponents, containing the values of $\frac{1}{\sqrt{x}}$, and another for odd exponents, which stores the values of $\frac{1}{\sqrt{2x}}$. Given that the normalised fraction range with the hidden bit 1 is defined as [1, 2), the reciprocal square root range for even exponents falls between [1, 0.7071...) and for odd exponents, it lies in the range [0.7071..., 0.5). Each table splits this range into 8 segments, providing initial approximations that are stored in an array [0:7], starting at index 0. It's crucial to acknowledge that the values in the lookup tables might be improper fractions, so they are truncated to the nearest 32-bit binary representation, although fewer bits could be employed to lessen resource constraints. The appropriate initial approximation is determined by the first three most significant bits of the number being squared, which serve as the index to access the array. For instance, if the fraction is 1.314 in decimal, with the fraction part excluding the hidden bit being 0.314, its 32-bit binary equivalent would be "01010000011000100100110111010010", and its initial three MSB bits, 010, would point to index 2. If the exponent is even, this index corresponds to approximately 0.8936 in the even lookup table, which is near the actual reciprocal square root of 1.314, approximately 0.8723. Thus, it becomes evident that the greater the number of segments, the more precise the initial approximation. In SystemVerilog, this algorithm can be executed using a mix of fixed-point multiplication and subtraction, in conjunction with lookup tables. It's noteworthy that each iteration requires three fixed-point multiplications, so a 96-bit array is utilised to store intermediate results, reducing rounding or truncation errors at the expense of greater computational resources.

While this approach produces satisfactory outcomes, it relies on fixed initial approximations determined by the number of segments. When dealing with large numbers, as indicated by a significant regime or exponent, precision can diminish, impacting the integer part of the calculation. Hence, additional refinements were attempted to improve the initial approximation and decrease the number of iterations needed. A simple initial piecewise reciprocal square root linear approximation algorithm was introduced prior to the application of the Newton-Raphson method as depicted in equation 3.14, aiming to increase precision.

$$f(x) = ax + b \tag{3.14}$$

This enhancement involves pre-calculating and storing the slope, a and intercept, b values for the piecewise function in the lookup tables, replacing the direct approximation values used previously. The division of the 8 segments is arranged as follows: $[x_1, x_2)$ includes [1, 1.125), [1.125, 1.25), [1.25, 1.375), [1.375, 1.5), [1.5, 1.625), [1.625, 1.75), [1.75, 1.875), and [1.875, 2). For each specified interval, a pair of equations 3.15 is formulated from chosen points:

$$ax_1 + b = \frac{1}{\sqrt{x_1}}, \; ax_2 + b = \frac{1}{\sqrt{x_2}} \tag{3.15}$$

Taking the earlier example of 1.314, which falls within the range of [1.25, 1.375), and setting $x_1 = 1.25$, $x_2 = 1.375$ in the simultaneous equations, a is approximately -0.333 and b is about 1.311. Hence, the initial reciprocal of square root approximation for 1.314 would be $(-0.333 \times 1.314) + 1.311 = 0.8734$, which is much closer to the actual value, 0.8723. It is important to note that these slope, a and intercept, b values are precalculated and stored in the lookup tables. The method of deriving the index to access the appropriate value in the tables is the same as previous.

Through these adjustments, the algorithm can produce accurate outcomes in two iterations. Yet, it was observed that its dependency on lookup tables and the necessity for three multiplications per iteration renders it inefficient concerning hardware resource utilisation.

In contrast to the Newton-Raphson method, the non-restoring algorithm, as detailed in reference [27] , is notable for its efficiency in hardware implementation. This efficiency stems from its reliance on fundamental binary operations, including shifting, addition, and subtraction. Unlike the straightforward application of the formula seen in the Newton-Raphson method, this version of the non-restoring algorithm involves a more intricate process, as elaborated and derived in the reference [27] as well. It meticulously manipulates bits of the partial quotient and partial remainder to calculate the square root pair by pair, as demonstrated in Algorithm 7.

For a 32-bit operand, the resulting square root will typically be represented in 16 bits since the algorithm computes one square root bit (quotient) for every two bits of the operand (1 pair), leading to 16 iterations. Diverging from conventional non-restoring algorithms, which adjust the partial root through addition or subtraction based on the result of D-Q×Q, the unique aspect of this algorithm is its consistent use of the partial remainder in subsequent iterations, irrespective of whether it's positive or negative. This approach forgoes restoring the remainder to its prior state even if it is negative, as seen in typical restoring algorithms.

Adjustments are necessary for compliance with the Posit floating-point square root format. The square root of the regime and exponent remains unchanged as described in Table 3.5. For the fraction part, it's extended to a 64-bit representation to achieve

Algorithm 7 Non-Restoring Square Root Algorithm

Input: 32-bit unsigned integer, D

Output: 16-bit unsigned integer, Q

```
$Q \leftarrow 0$                                        ▷ Quotient
$R \leftarrow 0$                                        ▷ signed integer, Remainder
for $i \leftarrow 15$ down to $0$ do
    $R \leftarrow (R << 2) \mid ((D >> (i << 1)) \,\&\, 3)$
    if $R \geq 0$ then
        $R \leftarrow R - ((Q << 2) \mid 1)$
    else
        $R \leftarrow R + ((Q << 2) \mid 3)$
    end if
    if $R \geq 0$ then
        $Q \leftarrow (Q << 1) \mid 1$
    else
        $Q \leftarrow (Q << 1) \mid 0$
    end if
end for
if $R < 0$ then
    $R \leftarrow R + ((Q << 1) \mid 1)$
end if
```

higher precision, necessitating 32 iterations. Given the fixed-point representation, the two most significant bits (MSB) of the 64-bit extension are allocated for the integer part. Allocating two bits for the integer part, rather than just a single bit, is a strategic choice that aligns with the operational mechanics of the algorithm. This is because the algorithm processes one pair of operand bits in each iteration. Additionally, for an odd exponent, the extended fraction is shifted left by one bit, effectively doubling it, aligning with the formula for fraction part when exponent is odd.

In summary, both methods are viable. The Newton-Raphson approach yields accurate results in fewer iterations but is more resource-intensive due to the use of lookup tables and three multiplications per iteration. Conversely, the non-restoring algorithm calculates the square root bits by bits, requiring more iterations but less hardware resources. Notably, most of the project time for this part was spent on research, functional implementation, debugging, and testing. Therefore, both codes focus primarily on functionality, with further optimisation needed to meet specific timing and hardware requirements.

3.1.4 Rounding

Different from IEEE-754 Floating Points, Posits only have one rounding mode, which is round half to even, where the unrepresentative (inexact) numbers are rounded to the nearest number. If the number is halfway between two others, it is rounded to the nearest even number. The process is detailed in Algorithm 8.

The rounding module takes a 64-bit fraction bit, a 2-bit exponent bit (E_O), and a 9-bit regime (R_O) from the arithmetic modules. The 64-bit fraction makes sure the result from the rounding module is correct. The module first combines the exponent and fraction bits (exp_frac_combine_output), because they will be rounded together depending on the size of the regime. The next step is to do a left shift operation, the MSB of the shifted "exp_frac_combine_output" becomes the LSB of the final output, as shown in Figure 3.2.

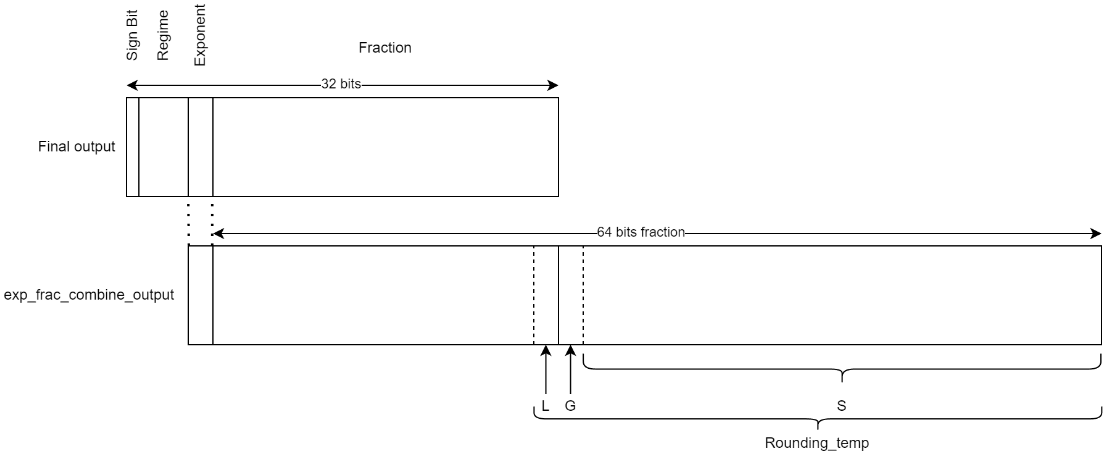


FIGURE 3.2: Rounding Method

It utilises three variables: L, the Least Significant Bit (LSB) of the number to be rounded; G, the Guard Bit, which is directly adjacent to L; and S, the Sticky Bit, which is the cumulative OR of all the bits beyond G. The LSB, L, determines the parity of the number, indicating evenness or oddness. The Guard Bit, G, together with the Sticky Bit, S, assesses the fractional part of the number relative to the midpoint of 0.5. A "round" indicator signals the necessity for rounding. According to the rules depicted in Table 3.6, the decision to round is based on the values of three indicators: L, G, and S. When both G and S are 0, or when G is 0 but S is 1, the value is considered less than halfway, and thus, no rounding is applied. If G is 1 and S is 0, the decision to round hinges on the Least Significant Bit (L). For values with L as 1, which indicates an odd number, a rounding step is performed to achieve evenness. Conversely, if L is 0, reflecting an even number, the value is already at the preferred even state, and no rounding is needed. In cases where both G and S are 1, the value exceeds the halfway point, and rounding is carried out regardless of L's state.

L	**G**	**S**	**round**
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1

TABLE 3.6: Rounding Rules

When doing multiplication and division instruction. The exponent is added together or subtracted from each other, which may produce a regime whose length is larger than 31 bits, which will cause problems when doing rounding since Posits never round to 0 or ∞. The value of R_O needs to be set to the correct value when the value is larger than the limit, as shown in Algorithm 8 lines 6 to 9.

Algorithm 8 Rounding

```
Input: Sign, Regime, Exponent, Fraction, Exact Exponent Sign, Zero and NaR flags
exp_frac_combine_output = {0, Exponent, Fraction}    ▷ concatenate exponent and fraction
rounding_temp = exp_frac_combine_output << (N-Regime-2)    ▷ Shift left according to the length of Regime
L = rounding_temp[MSB]
G = rounding_temp[MSB-1]
S = | (rounding_temp[MSB-2:0])    ▷ S is true if any of remaining bit is true
if Regime > 31 then
    rounding_condition = 0    ▷ regime is too large, no exponent and fraction bits
else
    rounding_condition = 1
end if
if (Regime > 31 && Exact Exponent sign is positive) then
    R_O_fin = 31    ▷ When regime is sequence of 1
else if (Regime > 31 && Exact Exponent sign is negative) then
    R_O_fin = 30    ▷ When regime is sequence of 0
else
    R_O_fin = Regime    ▷ normal case
end if
if (G && rounding_condition) then
    if S then    ▷ When number is larger than 0.5
        round = 1
    else
        round = L    ▷ when L is 1, odd number need rounding
    end if
else    ▷ When number is smaller than 0.5
    round = 0;
end if
exp_frac_output = exp_frac_combine_output[2*N+1:N+2] >> (R_O_fin+1)    ▷ Pick useful bits from exponent and fraction and shift to correct position
if (sign_Exponent_O) then    ▷ When exact exponent is negative
    regime_temp_output = 1 << (2*N-R_O_fin-2)
else    ▷ When exact exponent is positive
    regime_temp_output = ∼(1 << (2*N-R_O_fin-2))
end if
regime_output = regime_temp_output[2*N-1:N]    ▷ Pick useful bits from regime_temp_output
regime_output[N-1] = 1'b0    ▷ Keep sign bit of the output = 0 before handling sign
temp_output1 = (regime_output | exp_frac_output) + round    ▷ combine regime, exponent and fraction together and add round
if (Sign) then    ▷ negative number
    temp_output = -temp_output1    ▷ Change sign in 2's complement
else    ▷ positive number
    temp_output = temp_output1
end if
if (zero | NaR) then
    OUT = {NaR, {N-1{1'b0}}}    ▷ check Zero and NaR flag
else
    OUT = temp_output    ▷ normal output
end if
```

Utilising the previously calculated R_O_fin, the process of assembling the final output can commence. The initial step involves right-shifting the ”exp_frac_combine_output” to align the significant bits with the size of the sign together with the regime bits, resulting in the ”exp_frac_output”.

The regime output is then generated by either left-shifting 00...01 if the regime (negative) is a sequence of 0 with ending 1 or shifting 11...10 if the regime (positive) is a sequence of 1 ending by 0, as shown in Algorithm 8 lines 28 to 31. Subsequently, the most significant bit (MSB) is cleared to make room for the sign bit. This manipulation yields a temporary value composed exclusively of the regime bits, ”regime_output”, with the remaining bits set to zero. With ”exp_frac_output” already prepared from an earlier step, an OR operation can merge the ”regime_output” with ”exp_frac_output” and incorporate the rounding bit, as indicated Algorithm 8 line 35. In SystemVerilog, the inherent support for signed values allows for the straightforward setting of the sign bit with the output bits since Posit’s use of two’s complement for sign representation. The finalisation of the output also includes checks for zero and Not-a-Real (NaR) flags to determine if these special cases should be outputted.

Regarding the addition of the rounding bit to the final output, there is no need for the concern about fraction bit overflow due to rounding since such overflow occurs only when all fraction bits are set to ’1’. Typically, when the fraction and exponent bits are segmented, adding one to the fraction results in a carry that transforms the fraction to a sequence of zeros and increments the exponent by one. This carryover is naturally handled during the combination process.

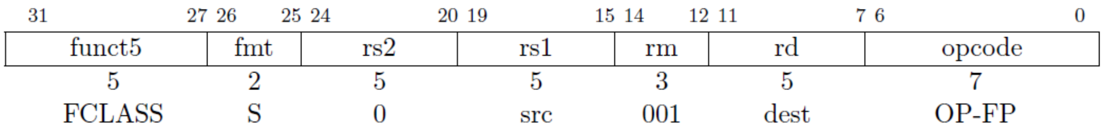
In situations where both the exponent and the fraction experience overflow, it will be accounted in the regime part. For a negative regime represented by a sequence of zeros ending in a ’1’, an overflow shifts this ’1’ to the left, effectively incrementing the negative regime. Conversely, for a positive regime characterised by a sequence of ones ending in a ’0’, adding one shifts this ’0’ to the right, with all other bits resetting to zero, which is consistent with the representation of an incremented positive regime.

3.1.5 Non-computational Logic

The non-computational block is distinct from prior arithmetic operations. It’s function is to execute simpler non-arithmetic instructions that include classification, comparison, finding minimum and maximum values, and sign injection. As with the previous two blocks, all the instructions implemented in the noncomp block focus on the single precision format with a bit length of 32.

3.1.5.1 Classifier

The FCLASS.S instruction (Figure 3.3) is utilised for classifying single-precision floating-point numbers. In the context of the IEEE-754 standard, this instruction inspects the value in the floating-point register rs1 and then outputs a 10-bit mask into the integer register rd. This mask is a binary representation where each bit corresponds to a specific class of floating-point number, including normal, subnormal, zero, infinity (each subdivided into positive and negative categories), as well as signaling NaN and quiet NaN. This detailed classification system helps in making precise mathematical distinctions and handling edge cases effectively.



31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCLASS	S	0	src	001	dest	OP-FP

FIGURE 3.3: Classification Instruction for FPU *(sourced from [1])*

On the other hand, the posit format simplifies this classification by only recognizing two categories: zero and the Not-a-Real (NaR) value. This simplification stems from the posit format's design philosophy, which aims to streamline the representation of numbers and reduce complexity. In posit format, zero is uniquely represented by a sequence of all 0 bits while NaR is represented by a leading most significant bit (MSB) followed by all zeros. Unlike IEEE-754, which accommodates a wide range of NaN values including signaling and quiet variants, the posit format's NaR serves as a singular representation for undefined or unrepresentable values, embodying a more straightforward error-handling approach. Hence, the 10-bit mask is reduced to a 4-bit mask as shown in Table 3.7.

Enumeration	**Description**
0001	Zero
0010	NaR
0100	Positive
1000	Negative

TABLE 3.7: Classification of Posit

In addition, the posit format diverges from IEEE-754 in its approach to handling numbers at the extremes. It does not employ subnormal numbers. Instead, it utilises tapered precision, where the precision of numbers decreases both as they become very small (approaching zero) and as they become very large. This symmetric treatment with tapered precision contrasts with IEEE-754, where subnormal numbers handle gradual underflow, but there is no symmetrical mechanism for gradual overflow in which large numbers could abruptly overflow to infinity.

The IEEE-754 format, with its detailed classification, allows for a more nuanced handling of special cases, which could be essential in some high-precision applications. In contrast, the posit format's simplified approach might offer advantages in terms of reduced computational overhead and simplicity, making it potentially more suitable for applications where hardware efficiency is a priority.

In short, while both IEEE-754 and posit formats have their unique approaches to classifying floating-point numbers, the choice between them depends on the specific requirements of the application, balancing the need for precision, computational efficiency, and simplicity in implementation.

The classifier implemented in this block is also used in other instruction handling, inlcuding both non-computational and arithmetic instructions, in order to ascertain whether certain inputs are any special values where they can then be handled correctly by using special case functions.

3.1.5.2 Comparison

In single-precision floating-point comparison instructions (Figure 3.4), two operands are designated as src2 and src1, corresponding to rs2 and rs1 register respectively. Three comparison instructions: equal, less-than, and less-than-and-equal are represented by rounding mode (rm) field in the instruction (although it is unrelated to rounding). The Floating-Point Unit FPU employs three specific instructions (FEQ.S, FLT.S, FLE.S) to perform comparisons between floating-point registers (rs1 $=$ rs2, rs1 $<$ rs2, rs1 $\leq$ rs2) where rs1 and rs2 are two input floating point numbers.

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCMP	S	src2	src1	EQ/LT/LE	dest	OP-FP

FIGURE 3.4: Comparison Instruction for FPU *(sourced from [1])*

Similarly, the Posit Processing Unit (PPU) implements these compare instructions for Posit representation inputs. As the compare instruction type is encoded within the rm field, existing numerators (RNE, RTZ, RDN) from the posit package (posit_pkg.sv) can be used to select the correct operation to be executed, where $RNE = LE$, $RTZ = LT$ and $RDN = EQ$.

It is noteworthy that for binary comparisons (yielding a result of 1 or 0), Posit numbers might be effectively compared as if they were signed integers. This is due to the similarity in how both Posit and signed integer formats utilise their most significant bit to indicate the sign: '0' for positive and '1' for negative values, as well as it does not have subnormal or duplicate representations. Such a setup ensures accurate comparisons between positive and negative Posits when treated as integers. In the case of negative Posits, which adopt the two's complement system just like negative signed integers, the integer representation of more negative values is numerically smaller. Consequently, when two negative Posits are compared as integers, the one with the lesser integer value correctly reflects being more negative. Conversely, in comparing two positive Posits, the one with a higher integer value represents the larger number, aligning with the comparison logic of unsigned integers. In scenarios where only the relative position (lesser, equal, or greater) between two numbers is relevant, without the need to quantify the magnitude of difference, this approach effectively yields a binary decision. Additionally, in the case of the NaR value, which is characterised by a leading '1' bit followed by all zeros, its two's complement form retains the same bit pattern. This means it could potentially be viewed as the "most negative number" within this system. Therefore, any negative number would be considered greater than the NaR value in such comparisons.

3.1.5.3 Min-Max

In the realm of floating-point arithmetic, instructions for finding the minimum (FMIN) and maximum (FMAX) operate by comparing two input floating-point numbers and writing either the smaller (for FMIN) or the larger (for FMAX) value into the destination register (rd). An important detail in these instructions is that the value -0.0 is regarded as smaller than +0.0. Moreover, the primary instruction for comparison remains FCMP, where rounding modes come into play to determine the specific operation (FMIN or FMAX).

In the context of Posit arithmetic, the Min-Max instruction serves the same comparable role, as described in section 3.1.5.2. These instructions ascertain either the smaller (Min) or larger (Max) value between two Posit numbers. The decision of which operation to execute (Min or Max) is guided by the rounding modes RNE and RTZ within the rounding module (roundmode_e) of the posit_pkg, with RNE corresponding to MIN and RTZ to MAX.

A unique feature of the Posit Min-Max instruction is how it handles inputs of Not a Real (NaR) value: if one of the input Posit numbers is NaR, the output of the Min-Max instruction will be the other, non-NaR input. Conversely, if both input Posit numbers are NaR, the output will also be NaR.

3.1.5.4 Sign injection

The sign-injection operations, as depicted in Figure 3.5 (FSGNJ.S, FSGNJN.S, and FSGNJX.S), primarily focus on manipulating the sign bit of the rs1 register while preserving its other bits. This functionality is similarly extended to posit numbers. However, as changing the sign of a floating point number essentially turns the value from a positive real to the same value negative real, whilst flipping the sign of a posit does not have the same effect, the function of the sign injection instructions were modified to mirror the functionality of the floating point implementation. The sign of the output would be calculated as seen in Table 3.8 and then compared with the sign of src1. Then if it differed the output would be the negated value of src1 giving the same posit value but with the sign changed.

The specific operation executed is determined by the rounding mode field in the instruction, where the enumerations give the operations as follows: $RNE = FSGNJ.S$, $RTZ = FSGNJN.S$, and $RDN = FSGNJX.S$.

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FSGNJ	S	src2	src1	J[N]/JX	dest	OP-FP

FIGURE 3.5: Sign-injection Instruction for FPU *(sourced from [1])*

Instruction	Posit 1	Posit 2	Sign of Output
SGNJ.S	src1	src2	Sign of src2
SGNJN.S	src1	src2	Negated sign of src2
SGNJX.S	src1	src2	XORs the sign of src1 and src2

TABLE 3.8: Summary of Sign-injection Operations for PPU

3.1.6 Conversion

In the Floating-Point Unit (FPU), the FCVT instructions are a crucial component of the conversion block, as depicted in Figure 3.6. These instructions facilitate the conversion between integer and single-precision floating-point formats, specifically through operations like FCVT.int.fmt and FCVT.fmt.int. All instructions within this block are coded under the OP-FP major opcode space. The FCVT.W.S and FCVT.L.S instructions, for instance, convert a floating-point number from the floating-point register (rs1) to a signed 32-bit or 64-bit integer, respectively, storing the result in the integer register (rd). In contrast, the FCVT.S.W and FCVT.S.L instructions perform the opposite function: they convert a 32-bit or 64-bit signed integer from the integer register (rs1) into a floating-point number, which is then stored in the floating-point register (rd). Additionally, the FCVT.WU.S, FCVT.S.WU, FCVT.LU.S, and FCVT.S.LU instructions handle conversions between floating-point numbers and unsigned integers.

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCVT.*int.fmt*	S	W[U]/L[U]	src	RM	dest	OP-FP
FCVT.*fmt.int*	S	W[U]/L[U]	src	RM	dest	OP-FP

FIGURE 3.6: Conversion Instruction for FPU *(sourced from [1])*

In the Posit-processing unit, the conversion block has been strategically redesigned and simplified, focusing exclusively on conversions between 32-bit signed or unsigned integers and posit numbers. This design choice resulted in the exclusion of 64-bit conversion operations (referred to as ’L’ instructions) present in the FPU’s conversion block. Notably, the FCVT.W.S and FCVT.WU.S instructions are used for converting a posit number into a 32-bit signed or unsigned integer, respectively. Conversely, FCVT.S.W and FCVT.S.WU instructions transform a 32-bit signed or unsigned integer into a posit number.

The subsequent sections detail the methodologies and algorithms for implementing these two core instructions within the conversion block. It's important to note that the development of the conversion block is not yet complete, as it was interrupted during the project's progression. A key unfinished aspect is confirming the length of each array, which requires extensive testing and verification to ensure both accuracy and functionality.

3.1.6.1 Posits to Integer

The conversion of Posit numbers to integers, both signed and unsigned, begins with an extraction process. This crucial initial step is elaborated upon in section 3.1.1 and further detailed in posit_extraction.sv. The extraction is essential for the subsequent conversion steps outlined in Algorithm 9.

The algorithm for converting posit to integer is detailed in Algorithm 9. It starts by computing the exact exponent 'EE', which is crucial for verifying the range of the input. This computation is based on the formula in Equation 3.16:

$$EE = \log_2((2^{2^{ES}})^r \times 2^e) = 2^{ES} \times r + e \tag{3.16}$$

This formula aids in ensuring the input falls within the permissible range. For the FCVT.W.S instruction, the valid input range is between -2^{31} and $2^{31} - 1$, while for FCVT.WU.S, the range is from 0 to $2^{32} - 1$. The comparison of the exact exponent 'EE' with these ranges is streamlined using a logarithmic approach. Per the posit standard specification, if 'EE' falls outside the acceptable range for either instruction, the resulting 32-bit integer will be formatted with the most significant bit (MSB) set to 1 and all other bits set to 0.

Algorithm 9 Conversion from Posit to integer

```
Input: 32-bit Posit,
Output: 32-bit Integer
Range Check: According to the type of instruction
Calculating the integer:
temp_num = $(-1)^{\text{Sign}} \cdot (2^{2^{\text{ES}}})^k \cdot 2^{\text{Exponent}} \cdot (1 + \text{Mantissa})$
Round the integer:
Shift the temp_num to the right for 32 bit to form a 64 bit number
if the 31th bit of the number is 1 then
    temp_num = temp_num + 1
else
    temp_num = temp_num
end if
Result integer:
int_output = temp_num
```

Algorithm 9 details the conversion process. It starts with a range check, followed by calculating the temporary number (temp_num) based on the sign, ES, regime (k), exponent, and mantissa. The integer is then rounded by shifting temp_num right for 32 bits to form a 64-bit number. If the 31st bit is 1, temp_num is incremented by 1; otherwise, it remains unchanged. The final step yields the integer output (int_output) from temp_num.

3.1.6.2 Integer to Posits

In the transition from integer to Posit formats, a unique case arises: an integer whose most significant bit (MSB) is set to 1 with all other bits being 0 is transformed into the NaR (Not a Real) Posit value. For all other integers that do not meet this specific criterion, the conversion adheres to the steps outlined in Algorithm 10. A critical aspect of this process is the handling of integers that effectively represent zero; in such cases, the resulting Posit is designated as Zero. This ensures a consistent translation from binary zero to a zero Posit value, maintaining uniformity across different numerical systems.

Algorithm 10 Conversion from integer to Posit

```
Input: 32-bit Integer,
Output: 32-bit Posit
Range Check: MSB_32bit = 32'b1000_0000_0000_0000_0000_0000_0000_0000
if input integer = MSB_32bit then
    posit output is NaR
end if
Find the leading bit position: leading_one_pos
Shift the input integer to the left by the length of leading bit position
Add_Mant_N = input_integer << shift_amount (shift to find out fraction part)
LE_O = N - leading_one_pos - 1;
E_O = LE_O[1:0];
R_O = LE_O[RS+ES:2];
Adapt the rounding module for the posit:
Calculating and rounding the posit number
Posit output after sign check: According to the type of instruction
For FCVT.S.W:
if sign 1 then
    posit_output = -temp_num
else
    posit_output = temp_num
end if
For FCVT.S.WU
posit_output = temp_num
```

The primary objective of this algorithm is to identify the leading bit's position in the integer, which aids in calculating the fractional part, regime, and exponent. The final posit number is then derived using the rounding principles detailed in Algorithm 8, as discussed in section 3.1.4. For sign determination, the MSB of the integer indicates the sign: a negative integer (sign 1) in FCVT.S.W leads to a negation of the temporary number (temp num), whereas a positive integer (sign 0) results in the direct use of temp num as the posit output. In the case of FCVT.S.WU, which converts unsigned integers, the posit output invariably mirrors temp num, as unsigned integers are always non-negative by definition.

3.2 PPU Integration

To integrate the PPU with the CV32E40p architecture, the existing open source FPU structure [24] serves as a foundational reference and model. However, certain complex features inherent in the FPU, specifically the pipeline, SIMD (Single Instruction Multiple Data) structure and PULP (Parallel Ultra Low Power) platform, have been excluded in this project to maintain simplicity. Furthermore, while the FPU is designed to support a variety of IEEE-754 floating point formats, including the FP8, FP16, FP32 and FP64, the scope of our current implementation is centred around 32-bit posit format exclusively. Hence, these advanced features and related control logics are ommitted.

3.2.1 Posit Package

A posit-based package, *posit_pkg.sv*, has been developed to facilitate the implementation of Posit Processing Unit (PPU) in SystemVerilog. This package is a centralised collection of user-defined data types, parameters, and functions specifically tailored for handling posit number systems. By encapsulating these elements into a single unit, the package ensures seamless accessibility and consistency across different modules in a SystemVerilog design environment.

The current package defines a single posit format, POSIT32, which is a 32-bit representation with a 2-bit exponent, as indicated by the *posit_format_e* enumeration and the *POSIT_ENCODINGS* parameter. The *POSIT_FORMAT_BITS* parameter, derived from the *NUM_POSIT_FORMATS*, facilitates the scalability of the package, allowing for the addition of new formats with minimal modifications. This design approach introduces flexibility, enabling the package to adapt to future requirements and advancements in posit number systems.

Furthermore, the *operation_e* and *opgroup_e* enumerations within the package play pivotal roles in defining and organising the implemented operations, as depicted in Table 3.9. The *operation_e* enumeration encompasses a comprehensive set of operations, including addition, subtraction, multiplication, division, square root, sign injection, min-max operations, comparisons, and classification.

Enumeration	Description	Enumeration	Description
0000	FMADD	0101	SQRT
0001	FMSUB	0110	SGNJ
0010	ADD	0111	MINMAX
0011	MUL	1000	CMP
0100	DIV	1001	CLASSIFY

Table 3.9: Enumeration and Corresponding Instructions for PPU Operations in cv32e40p Architecture

Concurrently, the *opgroup_e* enumeration further categorises these operations into four main groups: ADDMUL, DIVSQRT, NONCOMP, and CONV. Specifically, ADDMUL includes arithmetic operations like addition and multiplication, while DIVSQRT encompasses division and square root. NONCOMP is designated for non-computational tasks, including sign injection (SGNJ), min-max (MINMAX), comparisons (CMP), and classifications (CLASSIFY), and the CONV group is reserved for conversion tasks. This organisation offers a detailed representation of the package's operational capabilities but also promotes an orderly and effective methodology in managing posit number processing.

Finally, the package includes a set of helper functions that can be accessed by other modules. These functions are designed to assist with various tasks related to posit processing, such as determining the total posit size and exponent size, or performing other minor computational tasks specific to posit numbers. In essence, the package serves to define custom content in a well-organized manner, streamlining the integration and functionality of posit-related operations in different modules.

3.2.2 Hierarchy

Figure 3.7 presents a streamlined overview of how the PPU (Posit Processing Unit) is integrated into the cv32e40p architecture.

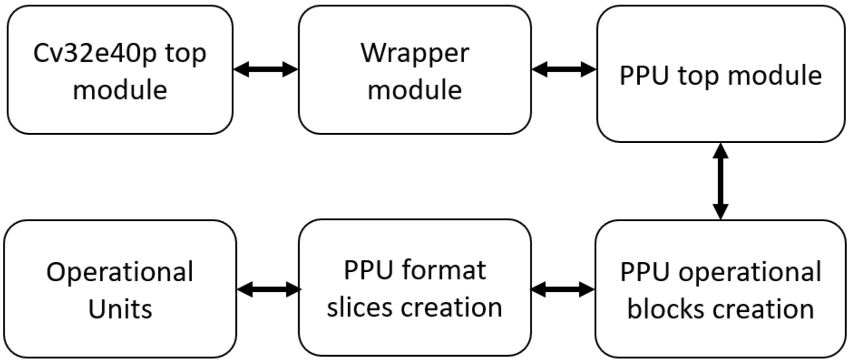


Figure 3.7: Hierarchical Structure of the PPU in the cv32e40p Architecture

The top-level module of cv32e40p, named cv32e40p_top, incorporates both the ppu_wrapper and the cv32e40p_core modules. The integration of the PPU is controlled by a parameter labeled "FPU," which can enable or disable the PPU functionality. Inside the ppu_wrapper module, the PPU's top-level module is initiated, leading to the formation of various operation groups. These groups are further categorised into specific formats. In this case, the implementation is confined to a 32-bit posit format slice. The most granular level comprises all operational unit modules, which include both arithmetic and non-computational modules. Table 3.10 provides a detailed list of the primary ports along with descriptions for the PPU.

Port Name	Direction	Type	Description
clk_i	in	logic	Clock
rst_i	in	logic	Reset
operands_i	in	logic [2:0][N-1:0]	Operands
rnd_mode_i	in	roundmode_e	Rounding mode
op_i	in	operation_e	Operation select
op_mod_i	in	logic	Operation modifier
result_o	out	logic [N-1:0]	Result
status_o	out	status_t	Status
in_valid_i	in	logic	Input data valid
in_ready_o	out	logic	Input interface ready
out_valid_o	out	logic	Output data valid
out_ready_i	in	logic	Output interface ready

Table 3.10: Primary Ports of PPU Top-level Module

3.2.3 Handshake Protocol

In the PPU, data flow is regulated through a synchronous valid/ready handshake mechanism, which is same with the protocol used in the existing FPU. This mechanism involves four primary signals: *in_valid_i*, *in_ready_o*, *out_valid_o*, and *out_ready_i*.

The process initiates with the assertion of the *in_valid_i* signal by an upstream module, such as the CPU, indicating that the incoming data is valid and stable. This signal stays asserted until the handshake is fully complete, ensuring the consistency and stability of the data throughout the process.

At the same time, the *in_ready_o* signal, indicative of a downstream module's readiness, signals the ability to process the data on the upcoming rising clock edge. The transaction reaches completion when both *in_valid_i* and *in_ready_o* are simultaneously asserted during a rising clock edge, marking a successful data transfer.

Once a transaction completes, the *in_valid_i* signal may remain active if there is additional data queued for the next operation, facilitating an efficient and ongoing process. Consistent with the existing FPU protocol, this handshake follows a top-down approach, meaning the state of the ready signals may depend on the valid signals, but the validity of the data is not contingent on the ready signals.

It is important to note that the roles of 'upstream' and 'downstream' are reversed when considering output signals, particularly after the data has been processed and is ready to be sent out. In this scenario, *out_valid_o* and *out_ready_i* perform functions opposite to *in_valid_i* and *in_ready_o*, respectively. The *out_valid_o* signal, maintained throughout the handshake, ensures the stability of outgoing data, while *out_ready_i* confirms the readiness of downstream interfaces to receive and process the data on the next clock edge.

Chapter 4: Testing and Evaluation

The modules of each operation unit within the Posit Processing Unit (PPU) undergo verification using Questasim UVM. Following this, they are joint into a comprehensive top-level test for the PPU. The final stage involves integrating these modules with the cv32e40p, facilitating the execution of tests written in C programming language.

4.1 Operation Units' Test

To test the accuracy of arithmetic units, a random number generator is written to generate test sequences in decimal as detailed in Appendix B.1. Besides, for accurate results, the test sequences are generated in double precision to avoid precision lost when being converted to Posit format. Following that, Softposit takes those test sequences, computing their theoretical outcome with corresponding functions and converting them into Posit format, as detailed in Appendix B.2. The converted test sequences are used by the testbenches, who takes two sequences as inputs to the computational units and the theoretical outcome sequence as the reference to the outcome from the units. The test sequences mainly focus on ± $(2^{-20}, 2^{16})$, which is the range where Posit can show its advantages in. Furthermore, the arithmetic units are also tested outside this range for uniform correctness. There are in total 1,200,000 test cases generated for each unit test where 400,000 of them are generated with in the range that Posit can harness its strength and other 800,000 are generated in other ranges. Bit pattern of the outcome is compared with the theoretical outcome converted with Softposit bit by bit to ensure the correctness.

4.2 PPU Top Level Test

Following the comprehensive testing of all instructions, the subsequent phase involves testing the entire Posit Processing Unit (PPU) at the top level. This step is to prepare for the eventual integration into the cv32e40p. A testbench has been developed to simulate the CPU signals as detailed in Table 3.10. For simulation purpose, the handshaking signals, *in_valid_i* and *out_ready_i* were fixed to 1, indicating the incoming data from the CPU is valid and the downstream components are ready for data processing. Ideally, the entire PPU structure would follow the layout depicted in Figure 3.1. However, the conversion block had been omitted, resulting in only three operational groups integrated: ADDMUL, DIVSQRT, and NONCOMP. Figure 4.1 displays the simulation results for the 18 instructions implemented. The primary goal of this test is not to perform exhaustive testing but rather to verify that the correct operations are executed where the CPU signals are correctly passed through and processed. Random operands are employed in this test to primarily assess the correct transmission and processing of signals from the CPU.

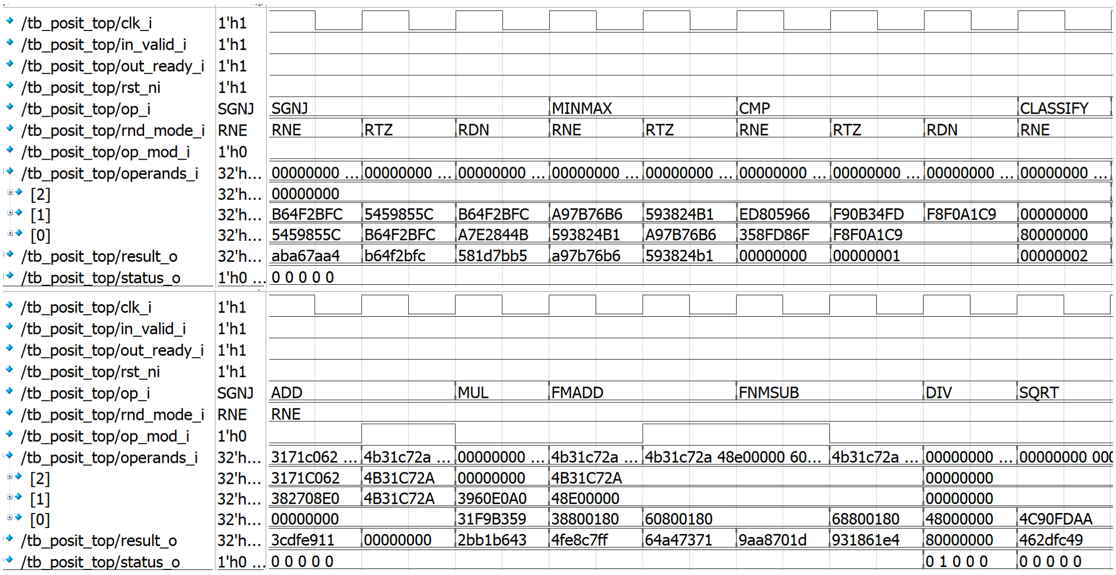

/tb_posit_top/clk_i 1'h1
/tb_posit_top/in_valid_i 1'h1
/tb_posit_top/out_ready_i 1'h1
/tb_posit_top/rst_ni 1'h1
/tb_posit_top/op_i SGNJ SGNJ MINMAX CMP CLASSIFY
/tb_posit_top/rnd_mode_i RNE RNE RTZ RDN RNE RTZ RNE RTZ RDN RNE
/tb_posit_top/op_mod_i 1'h0
/tb_posit_top/operands_i 32'h... 00000000 ... 00000000 ... 00000000 ... 00000000 ... 00000000 ... 00000000 ... 00000000 ... 00000000 ... 00000000 ...
[2] 32'h... 00000000
[1] 32'h... B64F2BFC 5459855C B64F2BFC A97B76B6 593824B1 ED805966 F90B34FD F8F0A1C9 00000000
[0] 32'h... 5459855C B64F2BFC A7E2844B 593824B1 A97B76B6 358FD86F F8F0A1C9 80000000
/tb_posit_top/result_o 32'h... aba67aa4 b64f2bfc 581d7bb5 a97b76b6 593824b1 00000000 00000001 00000002
/tb_posit_top/status_o 1'h0 ... 0 0 0 0 0
/tb_posit_top/clk_i 1'h1
/tb_posit_top/in_valid_i 1'h1
/tb_posit_top/out_ready_i 1'h1
/tb_posit_top/rst_ni 1'h1
/tb_posit_top/op_i SGNJ ADD MUL FMADD FNMSUB DIV SQRT
/tb_posit_top/rnd_mode_i RNE RNE
/tb_posit_top/op_mod_i 1'h0
/tb_posit_top/operands_i 32'h... 3171c062 ... 4b31c72a ... 00000000 ... 4b31c72a ... 4b31c72a 48e00000 60... 4b31c72a ... 00000000 ... 00000000 000
[2] 32'h... 3171C062 4B31C72A 00000000 4B31C72A 00000000
[1] 32'h... 382708E0 4B31C72A 3960E0A0 48E00000 00000000
[0] 32'h... 00000000 31F9B359 38800180 60800180 68800180 48000000 4C90FDAA
/tb_posit_top/result_o 32'h... 3cdfe911 00000000 2bb1b643 4fe8c7ff 64a47371 9aa8701d 931861e4 80000000 462dfc49
/tb_posit_top/status_o 1'h0 ... 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0

Figure 4.1: Testbench Results of Posit Top Module

Figure 4.1 demonstrates that the sign-injection, min-max finding, and comparison operations are correctly influenced by the rounding mode, aligning with the intended design. Furthermore, the signal *op_mod_i* plays its role in selecting instructions such as the FMSUB, FNMADD and SUB. The process of forming operation groups and the format slice is shown to be effective, as evidenced by the accurate and expected outcomes of each instruction. It is also noticeable that in cases of division by zero, the result is set to NaR, and the DZ status flag is raised.

Subsequent tests involved scenarios where one or both inputs were set to zero.

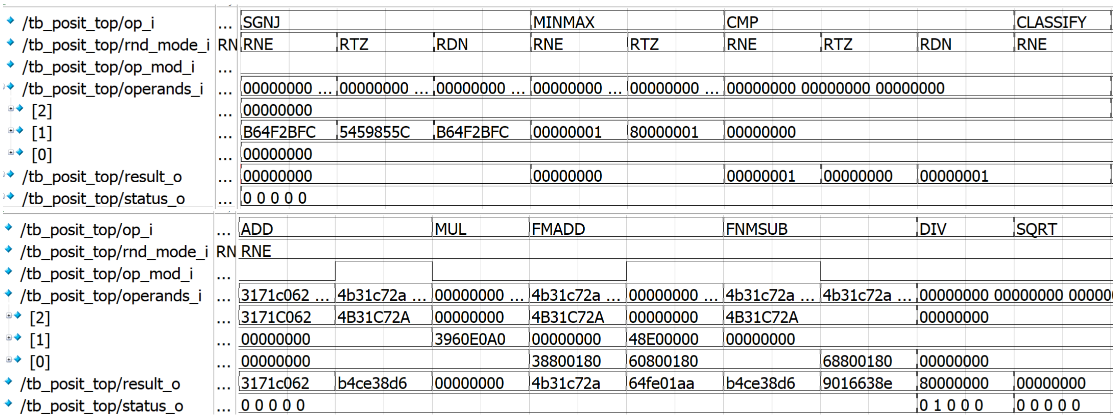

/tb_posit_top/op_i ... SGNJ MINMAX CMP CLASSIFY
/tb_posit_top/rnd_mode_i RN RNE RTZ RDN RNE RTZ RNE RTZ RDN RNE
/tb_posit_top/op_mod_i ...
/tb_posit_top/operands_i ... 00000000 ... 00000000 ... 00000000 ... 00000000 ... 00000000 ... 00000000 00000000 00000000
[2] ... 00000000
[1] ... B64F2BFC 5459855C B64F2BFC 00000001 80000001 00000000
[0] ... 00000000
/tb_posit_top/result_o ... 00000000 00000000 00000001 00000000 00000001
/tb_posit_top/status_o ... 0 0 0 0 0
/tb_posit_top/op_i ... ADD MUL FMADD FNMSUB DIV SQRT
/tb_posit_top/rnd_mode_i RN RNE
/tb_posit_top/op_mod_i ...
/tb_posit_top/operands_i ... 3171c062 ... 4b31c72a ... 00000000 ... 4b31c72a ... 00000000 ... 4b31c72a ... 4b31c72a ... 00000000 00000000 00000
[2] ... 3171C062 4B31C72A 00000000 4B31C72A 00000000 4B31C72A 00000000
[1] ... 00000000 3960E0A0 00000000 48E00000 00000000
[0] ... 00000000 38800180 60800180 68800180 00000000
/tb_posit_top/result_o ... 3171c062 b4ce38d6 00000000 4b31c72a 64fe01aa b4ce38d6 9016638e 80000000 00000000
/tb_posit_top/status_o ... 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0

Figure 4.2: Testbench Results with Input Zero

Figure 4.2 demonstrates that when the first operand in sign-injection operations is zero, the result is invariably zero, eliminating the possibility of a ’negative zero’. Moreover, the sign should not be inverted in such cases, as this would lead to a Not-a-Real (NaR) value. Tests involving boundary value comparisons, such as finding the minimum between zero and an extremely small number (7.52316384526264e-37), and the maximum between zero and a large negative number (-1.329227995784916e+36), both accurately resulted in zero. The correct classification of zero is further validated by the activation of the least significant bit in the mask bit. In addition, when performing addition, the non-zero positive value is simply passed through. Subtracting a positive value from zero yields its negative value. Multiplication by zero leads to a result of zero, while division by zero activates the DZ flag. It was also verified that zero divided by any number results in zero. Lastly, the square root of zero is confirmed to be zero as well.

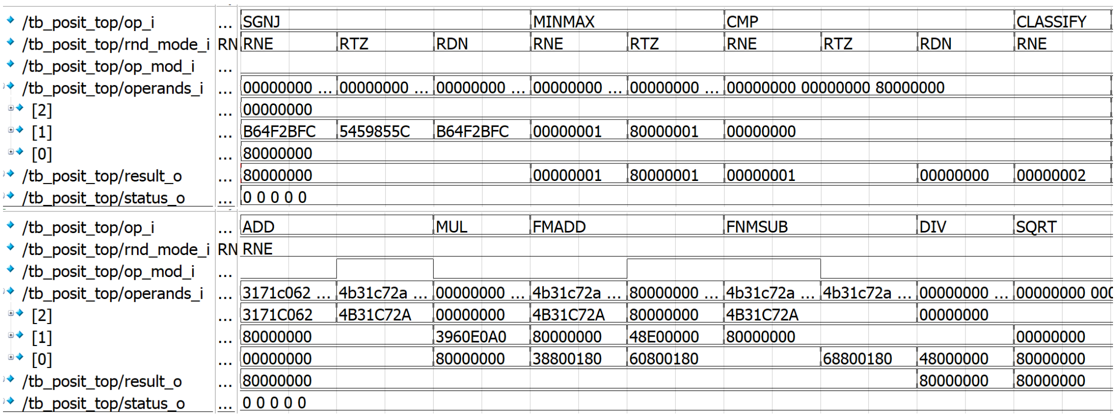

/tb_posit_top/op_i ... SGNJ MINMAX CMP CLASSIFY
/tb_posit_top/rnd_mode_i RN RNE RTZ RDN RNE RTZ RNE RTZ RDN RNE
/tb_posit_top/op_mod_i ...
/tb_posit_top/operands_i ... 00000000 ... 00000000 ... 00000000 ... 00000000 ... 00000000 ... 00000000 00000000 80000000
[2] ... 00000000
[1] ... B64F2BFC 5459855C B64F2BFC 00000001 80000001 00000000
[0] ... 80000000
/tb_posit_top/result_o ... 80000000 00000001 80000001 00000001 00000000 00000002
/tb_posit_top/status_o ... 0 0 0 0 0
/tb_posit_top/op_i ... ADD MUL FMADD FNMSUB DIV SQRT
/tb_posit_top/rnd_mode_i RN RNE
/tb_posit_top/op_mod_i ...
/tb_posit_top/operands_i ... 3171c062 ... 4b31c72a ... 00000000 ... 4b31c72a ... 80000000 ... 4b31c72a ... 4b31c72a ... 00000000 ... 00000000 000
[2] ... 3171C062 4B31C72A 00000000 4B31C72A 80000000 4B31C72A 00000000
[1] ... 80000000 3960E0A0 80000000 48E00000 80000000 00000000
[0] ... 00000000 80000000 38800180 60800180 68800180 48000000 80000000
/tb_posit_top/result_o ... 80000000 80000000 80000000
/tb_posit_top/status_o ... 0 0 0 0 0

Figure 4.3: Testbench Results with Input NaR

Figure 4.3 clearly shows that in sign-injection operations, if the first operand is NaR, the outcome is consistently NaR. The operation should not reverse the sign, resulting in a zero. In min and max operations, the returned value is simply the non-NaR operand. In terms of comparison for LE and LT, when NaR is the first operand, the result is 1, as NaR is considered less than any number. The classification accurately displays the correct bit mask, which is identified as the second bit. For all arithmetic operations, the presence of NaR as one of the inputs leads to NaR as the result.

4.3 C Programming Tests with cv32e40p

To test the successful integration of a Posit Processing Unit (PPU) into the cv32e40p, modifying the GCC compiler is essential to enable it to recognise and handle new data types for posits. This process involves adapting the compiler to parse these new types, perform accurate type checking, and, more critically, generate the correct Posit number representation. This means translating high-level C code into specific instructions that a posit-based FPU can understand, making the entire workflow from coding to execution ”posit-aware.”

Considering the limited timeframe of the project and the intricate task of modifying the GCC compiler from the ground up, particularly without previous experience, fully achieving this modification presents significant challenges. Consequently, two alternative approaches were devised to carry out the required testing without extensive alterations to the system. However, this limitation severely restricts the capability to develop sophisticated and effective test programs that could thoroughly investigate the advantages of the posit format.

Method 1: Manual Assembly Modification

This method involves compiling C code to assembly using the following command and then manually altering the assembly code. This command compiles C code for RISC-V systems using specific settings that cater to the 32-bit RISC-V architecture, including support for integer operations, single-precision floating-point operations, and compressed instructions. After compilation, the IEEE-754 floating-point values in the assembly code are manually replaced with their posit counterparts. This method leverages existing compiler infrastructure, avoiding modifications to the compiler or ISA, and offers direct control over machine instructions for implementing posit operations.

```
riscv32-corev-elf-gcc -march=rv32imfc_zicsr -mabi=ilp32 -S main.c
```

Method 2: Union-Based Representation in C

The second method uses a union in C, named PositBinary, to represent Posit numbers as shown in sample code 4.1. This union has two fields: a float and an unsigned integer. Posit values are input as unsigned integers using binary or hexadecimal representation, then interpreted as floating-point numbers during arithmetic operations. For example, assigning a posit value to the integer part of the union and then using it in a floating-point operation like result.f = a_temp.f * b_temp.f prompts the GCC to process it as a standard floating-point operation. The operands will be passed into the PPU as Posit binary representation.

```
union PositBinary {
  float p;
  unsigned int i;
};

union PositBinary a_temp, b_temp, c_temp, result;

printf("---------------- Posit FMADD Start ----------------\n");
// FMADD
for (int i = 0; i < 200; i++) {
    a_temp.i = test_a[i];
    b_temp.i = test_b[i];
    c_temp.i = test_c[i];
    __asm__ ("fmadd.s %0, %1, %2, %3" : "=f" (result.p) : "f" (a_temp.p),
    "f" (b_temp.p), "f" (c_temp.p));
    printf("%08x * %08x + %08x = %08x\n", a_temp.i, b_temp.i,
    c_temp.i, result.i);
}
printf("---------------- Posit FMADD End ----------------\n");
```

Listing 4.1: Posit FMADD code sample

Besides, inline assembly, accessed through ’__asm__” keyword in C could be utilised as a direct method to insert specific assembly instruction within the C code.

4.4 Performance and Benchmarking

It is acknowledged that the limitations imposed by the absence of a GCC compiler compatible with Posit arithmetic and the current range of operations implemented, the scope and depth of testing and performance benchmarking are constrained. Is is also important to note that the C programming codes developed are adjusted to the second workaround method. If a compatible GCC is available, the codes would need to be adjusted accordingly.

4.4.1 Quadratic Formula Testing

The quadratic formula, as shown in 4.1, stands as an important component in the fields of algebra and numerical analysis. This formula, representing a second-degree polynomial as $ax^2 + bx + c = 0$ could be used to compare various arithmetic systems, including IEEE-754 floating-point and Posit arithmetic, as discussed in [2]. Besides, this formula encompasses a variety of arithmetic operations, including addition, subtraction, multiplication, division, and square root, thus making it a comprehensive testbed for evaluating all implemented arithmetic modules. The C code is detailed in Appendix B.3. It could give valuable insights into the handling of rounding errors and other numerical challenges by each arithmetic standard. A crucial aspect of this evaluation involves the subtractive cancellation, a scenario where the outcome of an addition or subtraction is substantially smaller than the operands involved. This issue can be seen when b^2 substantially surpasses $4ac$, potentially leading to reduced precision in the computation of the discriminant and, consequently, impacting the accuracy of the resulting roots.

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{4.1}$$

Test Case 1:
Coefficients: $a = 1$, $b = 200$, $c = 3$
Discriminant: $\sqrt{b^2 - 4ac} = 199.969...$
Results:

- **Float:** Root 1 = -0.0149993896..., Root 2 = -199.9850006104...
- **Posit:** Root 1 = -0.0150012969..., Root 2 = -199.9850006103...

Test Case 2:
Coefficients: $a = 2$, $b = 100$, $c = 3$
Discriminant: $\sqrt{b^2 - 4ac} = 99.879...$
Results:

- **Float:** Root 1 = -0.0300178528..., Root 2 = -49.9699821472...
- **Posit:** Root 1 = -0.0300180912..., Root 2 = -49.9699821472...

Test Case 3:
Coefficients: $a = 3$, $b = 100$, $c = 2$
Discriminant: $\sqrt{b^2 - 4ac} = 99.879...$
Results:

- **Float:** Root 1 = -0.0200119019..., Root 2 = -33.3133201599...
- **Posit:** Root 1 = -0.0200120608..., Root 2 = -33.3133215904...

Using IEEE-754 64-bit double precision as a benchmark, the correctly computed digits are highlighted. Notably, Posit 32-bit arithmetic demonstrates slight better precision over IEEE-754 32-bit calculations in scenarios where the square root of the discriminant approximates the value of b closely.

4.4.2 Monte Carlo Pi Estimation

The Monte Carlo technique is a statistical method for approximating the value of π. It uses a straightforward geometric framework: a circle with a radius unit 0.5 is inscribed in a square with sides measure 1 unit each. The squares would be 1 unit, and the circle's are, calculated as πr^2 would be $\pi/4$. The circle's area to the square's area ratio also simplifies to $\pi/4$. This ratio is the key in estimating π by randomly generating points within the square and counting those that fall inside the circle as described in equation 4.2.

$$\pi \approx \frac{4 \times \text{Number of points inside the circle}}{\text{Total number of points}} \tag{4.2}$$

To initiate, random points within a unit square, ranging from [0,1] for both coordinates are generated. The algorithm detailed in Appendix B.4 checks if a point is within the circle based on the Pythagoras theorem condition $x^2 + y^2 \leq 1$, where x and y are the point's coordinates. π is then estimated by dividing the count of points inside the circle by the total number of points, and multiplying by 4.

The hypothesis was that the precision of the generated random points might impact the accuracy of the algorithm, particularly in the computation of the condition formula, which combines addition and multiplication operations that could introduce rounding errors. Over multiple iterations, these inaccuracies might accumulate. Given the range [0,1], 32-bit Posit, with its 27-bit precision compared to 32-bit IEEE-754, with its 23-bit precision was expected to estimate π more accurately and with more precision.

However, when tested, both the 32-bit Posit and IEEE-754 formats yield a similar pi value of approximately 3.120. . . , which still diverges from the true value. This discrepancy can be attributed to the limited sample size of 1000 points, which is inadequate for such as an estimation. Moreover, the distribution of the randomly generated values is also a crucial factor. In short, while the method is intriguing, a reassessment with a larger dataset is necessary to obtain more solid comparative results.

4.4.3 Sum of Reciprocal

In the assessment of 32-bit Posit against IEEE Floating Point, a specific performance test was employed, using the formula shown in equation 4.3. This test also included 64-bit double precision as a benchmark due to its 52-bit fraction bit, providing a more accurate reference than single precision and Posit.

$$x = \frac{1}{100 \times n + \pi} \tag{4.3}$$

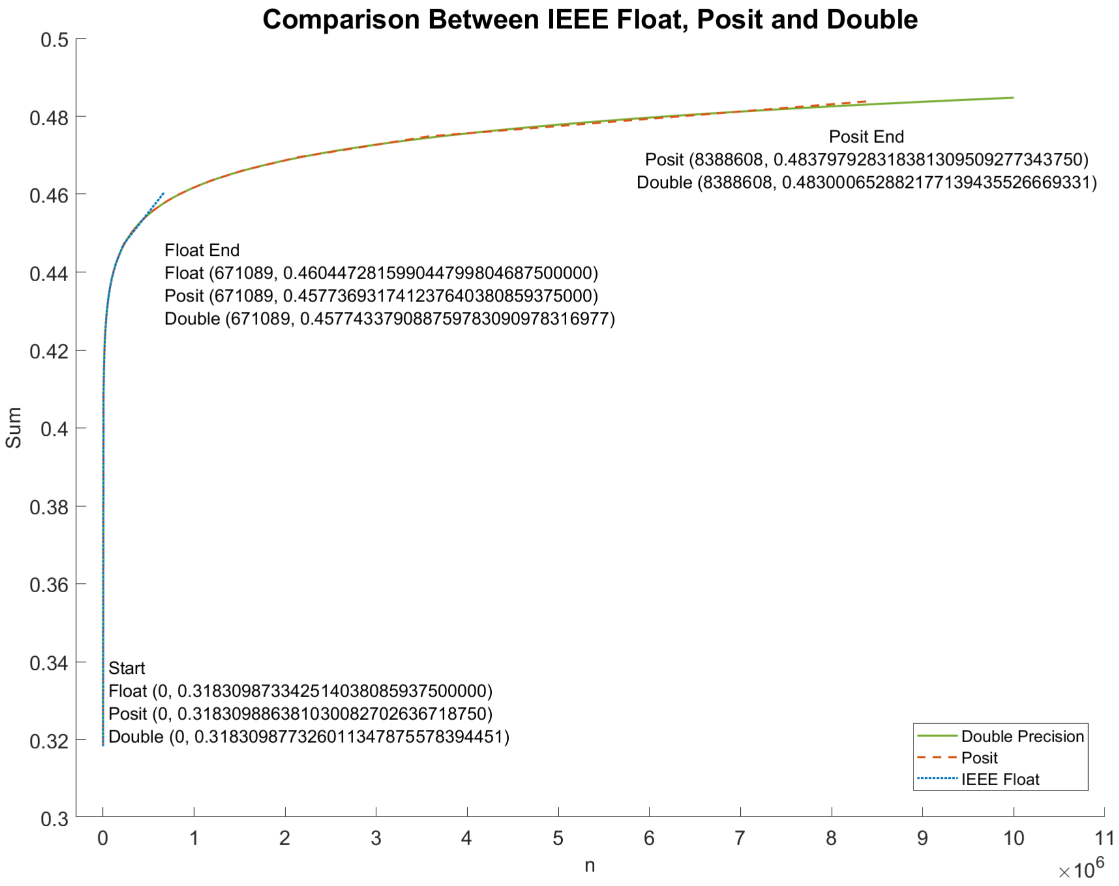
The test program involved summing up the values of x until reaching the precision limit of both IEEE Floating Point and Posit. Then, the reverse process was done to the sum. These limits were identified in two situations: first, when x became too small to be accurately represented, halting any further increase in the sum; and second, when n grew too large to be stored.

The initial results, as depicted in the figure 4.4, showed that both IEEE Float and Posit closely aligned with the reference line at smaller n values. However, the IEEE Float encountered its precision limit at n = 671089, with x becoming exceedingly small. This limit led to a significant deviation from the reference line in the final sum, as illustrated in Figure 4.5. In contrast, Posit continued to a much higher n value of 8388608, stopping only when n exceeded its storage capacity. Remarkably, even at this juncture, Posit's final sum remained closely aligned with the reference line, as shown in Figure 4.6. Furthermore, both formats underwent a reverse calculation process, retracing their steps back to the initial values.

The error rate for x in equation 4.3 is determined by dividing the difference between the sums obtained using IEEE Float and Posit from the reference sum computed in Double Precision, as outlined in equation 4.4. When comparing with double precision, the average error rate observed for IEEE Float in computing x is 0.5907%. In contrast, Posit exhibits a significantly lower error rate of 0.00140851% for the same number of n iterations. At the end of the calculations, Posit's error rate stands at 0.16506715%. These results indicate that Posit, on average, yields smaller errors compared to IEEE Float. Posit is better particularly at handling value near 2^0, where in this case, only 5 bits are allocated to sign, regime and exponent, leaving a larger fraction bits than the IEEE Float. This allocation results in enhanced precision for Posit over IEEE Floating Point format under these specific conditions.

$$error\ rate = \frac{|Posit\ or\ Float\ Final\ Value - Double\ Precision\ Value|}{Double\ Precision\ Value} \tag{4.4}$$

Although Posit shows advantages in some cases, it is not universally superior to IEEE Float. Specifically, the largest integer that can be represented accurately in this specific experiment without losing precision is 16,777,216 for IEEE Float, whereas for Posit, this number is only 8,388,608. Beyond these values, no bits are available to represent digits after the decimal point. In general, when dealing with numbers larger than 2^{20}, Posit has fewer fraction bits compared to IEEE Floating Point. This characteristic implies that IEEE Float is more suitable for computations involving extreme numbers where maintaining precision for decimal places is crucial. In such scenarios, Posit begins to lose its edge over IEEE Float due to its bit allocation scheme.


Comparison Between IEEE Float, Posit and Double

Posit End
Posit (8388608, 0.483797928318381309509277343750)
Double (8388608, 0.483000652882177139435526669331)

Float End
Float (671089, 0.460447281599044799804687500000)
Posit (671089, 0.457736931741237640380859375000)
Double (671089, 0.457743379088759783090978316977)

Start
Float (0, 0.318309873342514038085937500000)
Posit (0, 0.318309886381030082702636718750)
Double (0, 0.318309877326011347875578394451)

Figure 4.4: Comparison between IEEE Float, Posit and Double

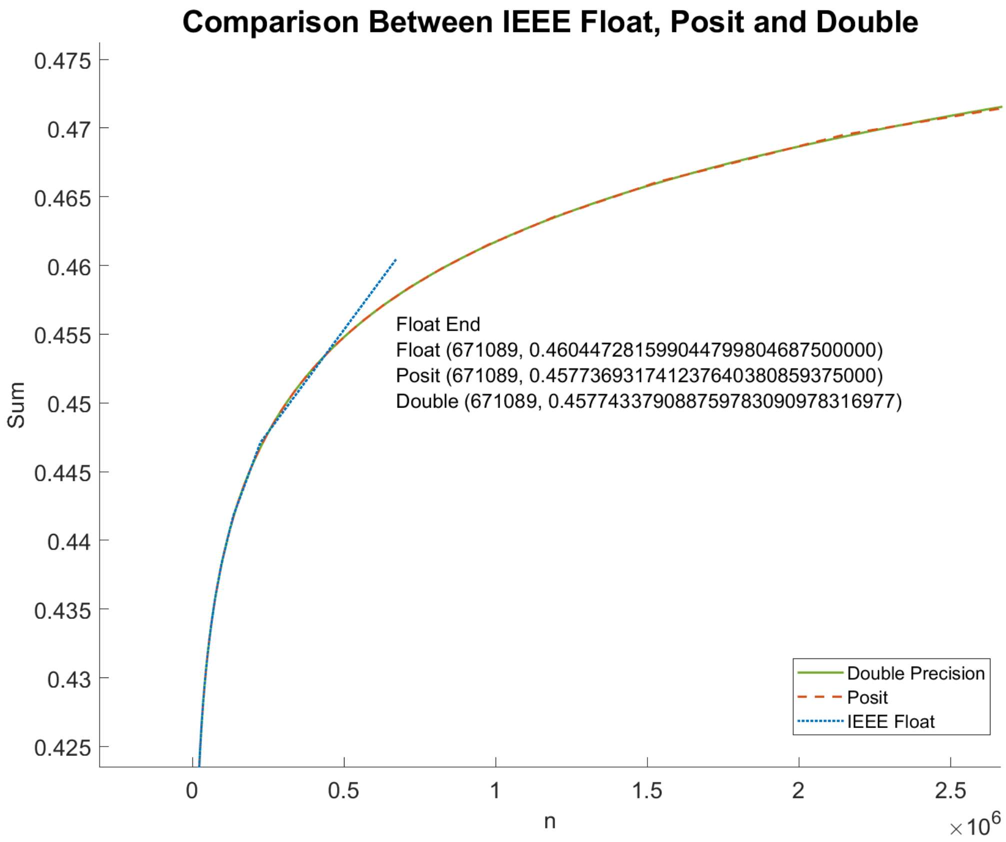


FIGURE 4.5: Comparison between IEEE Float, Posit and Double (Float End)

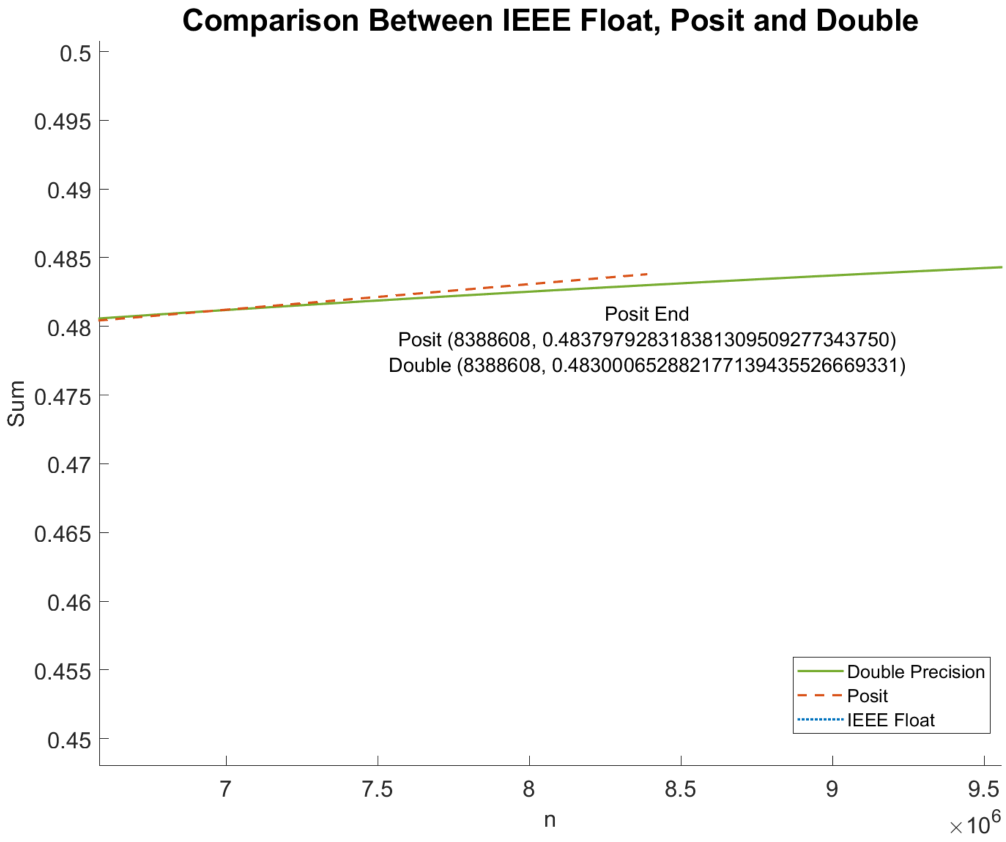


FIGURE 4.6: Comparison between IEEE Float, Posit and Double (Posit End)

Chapter 6: Conclusion and Future Work

6.1 Conclusion

The project reached a milestone with the successful adaptation of 18 instructions to the Posit format, in line with the RISC-V “F” standard extension. These included 9 arithmetic and 9 non-computational or logical instructions. The Posit Processing Unit (PPU), mirroring the structure of the cv32e40p’s existing IEEE-754 based Floating-Point Unit (FPU), was integrated successfully with cv32e40p at software level. This integration encompassed three out of four operational groups – ADDMUL, DIVSQRT, and NONCOMP, with CONV group being omitted due to time constraints. To access the functionality, various C programming test were developed, providing an initial exploration into the Posit standard’s benefits, particularly its tapered prevision. These tests suggested a potential increase in accuracy within the Posit’s sweet spot range compared to traditional formats. However, the project’s scope was limited by the time-intensive nature of modifying instructions and the lack of full GCC compiler compatibility, which restricted to simpler testing scenarios. Despite these challenges, the project yielded valuable insights into the Posit standard’s streamlined design and efficiency. A notable observation was the Posit format’s tapered prevision, which allows for more effective bit utilisation, reducing redundancy common in other formats. Additionally, the absence of subnormal values and simplified classification of special values potentially point toward a reduced need for complex control logic in computational processes. As the project concludes, it is evident that while substantial progress in developing a functional PPU was achieved, the exploration of Posit’s full capabilities remains an area for future exploration. The insights and experience gained lay a solid groundwork for future advancements in this promising field, especially in harnessing Posit’s potential for enhanced computational accuracy and efficiency.

6.2 Future work

The produced work can be furthered and improved upon by potential future projects or external open-source efforts.

6.2.1 Optimisation

The current implementation of the system's modules relies exclusively on combinational logic, which, while theoretically functional, poses challenges for practical hardware deployment due to timing and resource constraints. To enhance the design for real-world applications, especially for high-frequency operations, several optimisations are proposed:

1. Sequential Logic for DIVSQRT:

The DIVSQRT operational block is responsible for executing division and square root arithmetic. Its intricate operations are too complex for a single clock cycle. To manage these operations effectively, a multi-cycle, sequential method is required, allowing the division of tasks into smaller, more manageable parts that can be processed over multiple clock cycles. As a result, there is also a need to develop a more robust control signal framework. This is particularly crucial when integrating with the CV32E40P, as it ensure the precise timing and synchronisation of handshaking signals, which is essential for maintaining data integrity and system stability.

2. Pipelining Integration:

The introduction of pipelining into the PPU modules is anticipated to enhance throughput and support operation at higher clock frequency. By segmenting the processing flow into distinct stages, each separated by registers, the system can handle multiple instructions concurrently, each in different phases of processing. Given that the current FPU within the CV32E40P is already designed with a pipelined architecture, this existing framework can serve as a blueprint for extending pipelining to the PPU modules. This strategy promises a smoother transition and potential performance gains by leveraging the established pipelining principles.

3. Efficient Arithmetic Algorithms:

The current approach to multiplication and division, which utilises fixed-point arithmetic for fraction parts, should be re-evaluated. More hardware-efficient algorithms are recommended for adoption. For multiplication, methods such as Booth's algorithm or Wallace trees could be explored for their compactness and speed. Meanwhile, division could benefit from the implementation of a non-restoring division algorithm, known for its simplicity and efficiency. Additionally, the implementation of the non-restoring algorithm for the square roots could be structured around a Controlled Add-Subtract (CAS) architecture as proposed in reference [16], which promotes a more hardware-friendly circuit design for square roots.

4. Completion of Conversion Instructions:

The current version of the PPU has excluded the conversion operation group and its associated instructions, converting from Posit to integer and vice versa, due to strategic decisions to drop these tasks because of time limitations. Therefore, to ensure the PPU is fully functional and complete, these conversion instructions should be developed and their accurate functionality verified.

6.2.2 Hardware Synthesis and Implementation

The transition from functional testing to hardware implementation is a critical phase of the development process. While our designs have proven functionally sound in simulations, synthesis represents the next pivotal step. Synthesis involves transforming high-level designs into actual hardware representations, consisting of gates and connections tailored for a specific FPGA. This phase is far from trivial; it demands a thorough understanding of the design's practical implications, including considerations for area, power, and timing requirements, as well as temperature constraints.

Particularly, the timing analysis is of utmost important to confirm that the design adheres to the desired clock frequency specifications. Power analysis is equally crucial to guarantee energy efficiency, an essential attribute for hardware operation cost-effectiveness. This aspect becomes even more vital given the project's emphasis on posit arithmetic, which promises a simpler architecture, potentially contributing to reduced power consumption, aligning with the project's goals of creating an efficient and economical hardware solution. Setting up synthesis constraints is also a part of this process, including configuring the clock and various I/O ports to ensure smooth operation and to uncover issues like race conditions, glitches, inferred latches, setup and hold time violations that may not manifest during functional simulation.

Efficient utilisation of FPGA resources is another key focus area. It involves optimising the design to use Look-Up Tables (LUTs), flip-flops, and block RAM effectively, ensuring the best possible performance from the FPGA. The design's journey from a simulated environment to an actual FPGA board will also include physical design tasks such as floorplanning, placement, and routing—each critical for achieving an optimal layout that meets the design's requirements.

If the intended target board is the Nexys-A7 100T, as outlined in the stretch goals, or a more powerful board like the Genesys series, the Vivado Design Suite offers comprehensive tools for all design stages of FPGA, from functional simulation to synthesis, implementation, and bitstream generation. Additionally, basic guidelines for synthesizing the cv32e40p are available in reference [13]. It requires clock gating cells, which are typically specific to the selected FPGA technology. Although a simulation-only version of these cells is available, they are not suitable for synthesis. Hence, by addressing these aspects of design's post-functional testing, the conceptual simulation of our project can be advanced to a promising practical hardware product.

6.2.3 GCC Compiler

Given the project's limited time-frame and the lack of experience in compiler modifications, successfully adapting the GCC compiler for Posit support posed a significant challenge. In its default configuration, the RISC-V GCC compiles applications containing floating-point literals into the IEEE-754 format, following the specifications of the RISC-V "F" extension. For the current tests involving the cv32e40p and the Posit Processing Unit (PPU), less efficient methods are resorted. These included manually compiling to assembly code and altering IEEE-754 values into Posit format, or using a union in C to assign values to a floating-point register in the cv32e40p. These workaround methods, while functioning, proved to be inefficient and inconvenient, limiting the ability to run complex tests or algorithms.

Furthermore, special values like NaN (Not a Number) present additional complications. The definition and handling of NaN in IEEE-754 floating-point format differ from Posit format, leading to potential errors or undefined operations when encountering such values. To truly support Posit and enable more sophisticated testing and performance evaluation, the need to modify the GCC compiler is recognised. This modification is intended for generating Posit representations instead of IEEE-754 ones, leveraging the existing "F" extension of RISC-V but tailored for the PPU. In short, implementing these changes is essential for expanding the testing capabilities beyond simpler tests and accurately assessing the performance of the Posit format.

6.2.4 Performance Evaluation

Should more time be available, a thorough performance evaluation of this project would ideally encompass two key areas: software and hardware advantages.

1. Software:

Currently, the inability to fully evaluate the system's performance stems from two primary limitations: the absence of a GCC compiler compatible with Posit arithmetic and the restriction of the implementation of basic arithmetic operations, including addition, subtraction, multiplication, division, and square root given time limit.

As a result, the ability to test more complex functions is restricted, such as those involving trigonometry, logarithms, exponential, and power calculations. Consequently, The performance of Posit standard on advanced mathematical equations or algorithms could not be assessed, which would typically include power series, Taylor series, Fourier series, and various machine learning algorithms. The comprehensive evaluation of Posit's advantages, particularly its superior performance in ranges close to 1, its handling of special values, and the elimination of overflow and underflow issues, as well as reduced rounding errors, remains as another new journey.

Hence, to overcome these challenges and unlock the full potential of Posit, future work must focus on resolving the compiler compatibility issue and expanding the functionality of the current implementation. By addressing these areas, a more detailed and nuanced evaluation of Posit's benefits can be conducted. This will not only illustrate the strengths of Posit in computational accuracy and efficiency but also pave the way for its application in more complex mathematical and algorithmic domains.

2. Hardware:

In the course of implementing the Posit standard within this project, several key observations have been made concerning its hardware implications. A notable aspect of the Posit format is its limited number of special values, comprising only zero and NaR, which stands in contrast to the IEEE-754 standard that encompasses a wider range of special values and the presence of subnormal numbers. Furthermore, Posit simplifies the rounding process by offering just one mode, as opposed to the multiple rounding modes present in IEEE-754. These features of the Posit standard imply a potential reduction in the complexity of RTL (Register-Transfer Level) logic, which could lead to more efficient hardware designs.

However, there are trade-offs to consider. The format's dynamic bit allocation introduces a need for additional logic in the extraction module, as it requires the size of each Posit component – the regime, exponent, and fraction to be dynamically determined during runtime. This aspect adds a layer of complexity that requires further investigation to assess whether the Posit standard truly enhances hardware utilisation and efficiency.

Another intriguing aspect of the posit format is its tapered precision. This characteristic implies that algorithms optimised for the 'sweet spot' of posit arithmetic might require smaller bit sizes compared to equivalent floating-point operations while still achieving equal or superior accuracy. Such a feature could translate into reduced hardware requirements. Investigating how the tapered precision of posits can be used in specific algorithms could lead to opportunities for achieving high computational accuracy with lower hardware resource consumption.

Bibliography

[1] A. Waterman, K. Asanovic, "The risc-v instruction set manual," 2017. [Online]. Available: https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf

[2] J. L. Gustafson and I. T. Yonemoto, "Beating floating point at its own game: Posit arithmetic," superfri, vol. 4, no. 2, pp. 71–86, 2017. [Online]. Available: https://doi.org/10.14529/jsfi170206

[3] Z. Carmichael, S. H. Fatemi Langroudi, C. Khazanov, J. Lillie, J. Gustafson, and D. Kudithipudi, "Deep positron: A deep neural network using the posit number system," 12 2018. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8715262

[4] J. Lu, C. Fang, M. Xu, J. Lin, and Z. Wang, "Evaluations on deep neural networks training using posit number system," IEEE Transactions on Computers, vol. 70, no. 2, pp. 174–187, 2021. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9066876

[5] Embecosm Ltd., "About us," 2020. [Online]. Available: https://www.embecosm.com/about/

[6] OpenHW Group, "Explore our members." [Online]. Available: https://www.openhwgroup.org/membership/members/

[7] DIGILENT, "Nexys a7™ fpga board reference manual," 2019. [Online]. Available: https://digilent.com/reference/_media/reference/programmable-logic/nexys-a7/nexys-a7_rm.pdf

[8] "IEEE Standard for Floating-Point Arithmetic," IEEE Std 754-2019 (Revision of IEEE 754-2008), pp. 1–84, 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8766229

[9] J. L. Gustafson, "Posit arithmetic," 2017. [Online]. Available: https://posithub.org/docs/Posits4.pdf

[10] A. Cuyt, P. Kuterna, B. Verdonk, et al., "Underflow revisited," CALCOLO, vol. 39, pp. 169–179, 2002. [Online]. Available: https://link.springer.com/article/10.1007/s100920200003

[11] Posit Working Group, "Standard for posit arithmetic (2022)," 2022. [Online]. Available: https://posithub.org/docs/posit_standard-2.pdf

[12] s. H. Leong, “Introduction to softposit,” 2018. [Online]. Available: https://posithub.org/docs/PositTutorial_Part1.html

[13] OpenHW Group, “Cv32e40p user manual,” 2023. [Online]. Available: https://docs.openhwgroup.org/projects/cv32e40p-user-manual/en/latest/

[14] I. Yonemoto, “Sigmoid numbers for julia.” 2018. [Online]. Available: https://github.com/interplanetary-robot/SigmoidNumbers

[15] K. Nouh and H. A. H. Fahmy, “Binary floating point verification using random test vector generation based on sv constraints,” pp. 433–436, 2015. [Online]. Available: https://ieeexplore.ieee.org/document/7440341

[16] S. Samavi, A. Sadrabadi, and A. Fanian, “Modular array structure for non-restoring square root circuit,” pp. 957—966, 2008. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762108000623

[17] H. Zhang and S.-B. Ko, “Design of power efficient posit multiplier,” IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 67, no. 5, pp. 861–865, 2020. [Online]. Available: https://ieeexplore.ieee.org/document/9035440

[18] S. Tiwari, N. Gala, C. Rebeiro, and V. Kamakoti, “Peri: A posit enabled risc-v core,” 2019. [Online]. Available: https://arxiv.org/abs/1908.01466

[19] M. K. Jaiswal and H. K.-H. So, “Universal number posit arithmetic generator on fpga,” in 2018 Design, Automation Test in Europe Conference Exhibition (DATE), 2018, pp. 1159–1162. [Online]. Available: https://ieeexplore.ieee.org/document/8342187

[20] H. Jaiswal, Manish So, “Architecture generator for type-3 unum posit adder/subtractor,” pp. 1–5, 2018. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8351142

[21] R. Chaurasiya, J. Gustafson, R. Shrestha, J. Neudorfer, S. Nambiar, K. Niyogi, F. Merchant, and R. Leupers, “Parameterized posit arithmetic hardware generator,” in 2018 IEEE 36th International Conference on Computer Design (ICCD), 2018, pp. 334–341. [Online]. Available: https://ieeexplore.ieee.org/document/8615707

[22] J. Lu, S. Lu, Z. Wang, C. Fang, J. Lin, Z. Wang, and L. Du, “Training deep neural networks using posit number system,” in 2019 32nd IEEE International System-on-Chip Conference (SOCC), 2019, pp. 62–67. [Online]. Available: https://ieeexplore.ieee.org/document/9088105

[23] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. Gurkaynak, and L. Benini, “Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices,” Feb. 2017. [Online]. Available: https://ieeexplore.ieee.org/document/7864441

[24] S. Mach, F. Schuiki, F. Zaruba, and L. Benini, “Fpnew: An open-source multiformat floating-point unit architecture for energy-proportional transprecision computing,” IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 29, no. 4, pp. 774–787, 2020. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9311229

[25] S.G. Johnson, “Square roots via newton’s method,” 2015. [Online]. Available: https://math.mit.edu/~stevenj/18.335/newton-sqrt.pdf

[26] M. Morris, “Computing fixed-point square roots and their reciprocals using goldschmidt algorithm,” 2020. [Online]. Available: https://www.fpgarelated.com/showarticle/1347.php

[27] Y. Li, W. Chu, “A new non-restoring square root algorithm and its vlsi implementations,” 1996. [Online]. Available: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=0897444f6814a52978f4f23f5eb7db394a27727a

Appendix A: Electronic Appendix Index

- **materials (folder)**
  - **src (folder)** : RTL codes
    - posit_pkg.sv
    - posit_top.sv
    - posit_opgroup_block.sv
    - posit_opgroup_fmt_slice.sv
    - posit_extraction.sv
    - posit_LB_detector.sv
    - posit_fma.sv
    - posit_divsqrt.sv
    - posit_div.sv
    - posit_sqrt.sv
    - posit_sqrt_NR.sv
    - posit_noncomp.sv
    - posit_classifier.sv
    - posit_rounding.sv
    - cf_math_pkg.sv
    - lzc.sv
    - rr_arb_tree.sv
    - registers.svh
  - **test (folder)**
    - **c_test (folder)**: Custom C test programs
      - Float_and_Double_Test
      - basic.c
      - estimate_pi.c
      - mult.c
      - quadratic.c
      - reciprocal.c
      - squareAdd.c
      - testcase.h
      - randomValue.h

        - **README.txt**
    - **uvm_test (folder)** : Testbenches for unit tests
        - tb_posit.sv
        - tb_sqrt.sv
        - Posit_Divider_32bits_tb.sv
        - FMA_32bits_tb.sv
- **weekly reports**
    - Posit Team Weekly Report 30 Oct.docx
    - Posit Team Weekly Report 06 Nov.docx
    - 13_11.docx
    - Posit Weekly Report 19 Nov.docx
    - Posit Weekly Report 27 Nov.docx
    - Posit Weekly Report 04 Dec.docx
    - Posit Weekly Report 11 Dec.docx
- cv32e40p_fp_wrapper.sv
- Makefile

Appendix B: C Codes

B.1 Random Number Generator for Testing Sequences

```
// Start of Random Number Generator for Testing Sequences
#include "source/include/softposit.h"
#include <iostream>
#include <random>
#include <cstdlib>
#include <time.h>
#include <set>
#include <iomanip>
#include <windows.h>
#include <chrono>
#include <cmath>

using namespace std;

double rand_gen(double min, double max);

int main()
{
    double power;
    std::set<int> rand_power;
    for(int i = -45; i <= 36;)
    // for (int i = 0; i < 21;) //210k
    {
        // Generate random power
        int temp = distribution(generator);

        // if not in rand_power list
        if (rand_power.find(temp) == rand_power.end())
        {
            rand_power.insert(temp);
            power = pow(10, temp);
            // cout << power << endl;
            rand_gen(-1.33 * power, 1.33 * power);
            i++;
        }
    }
    return 0;
}

double rand_gen(double min, double max)
{
    // std::random_device rd;
    auto currentTime = std::chrono::system_clock::now();

    // convert to ms
    auto milliseconds = std::chrono::duration_cast<std::chrono::milliseconds>
    (currentTime.time_since_epoch());

    // get ms as seed
    unsigned long seed = static_cast<unsigned long>(milliseconds.count());

    std::default_random_engine generator(seed);
    std::uniform_real_distribution<double> distribution(min, max); // (0, 1]

    std::set<double> uniqueNumbers; // save unique rand numbers

    posit32_t p1, p2;
    double random_value, temp;
    // Generage double in [min, max]
    int count = 1;
    for (int i = 0; i < 100;)
    {
        random_value = distribution(generator);
        if (uniqueNumbers.find(random_value) == uniqueNumbers.end())
        {
            uniqueNumbers.insert(random_value);
            p1 = convertDoubleToP32(random_value);
            std::cout << std::showpoint << std::fixed << std::setprecision(55)
            << ": " << random_value << ",";
            printBinary((uint64_t *)&p1.v, 32);
            i++;
            Sleep(10);
        }
    }
    // double randomValue = distribution(generator);

    return 0;
}
// End of Random Number Generator for Testing Sequences
```

Listing B.1: random generator for testing sequences

B.2 Arithmetic Computation Using Softposit

```
// Start of Arithmetic Computation using SoftPosit
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <bitset>
#include <windows.h>

#include "source/include/softposit.h"

int main()
{
    std::ifstream file_in1("full_range_in1.txt");
    std::ifstream file_in2("full_range_in2.txt");

    if (!file_in1.is_open())
    {
        std::cerr << "Cannot open in1_fin" << std::endl;
        return 1;
    }

    if (!file_in2.is_open())
    {
        std::cerr << "Cannot open in2_fin" << std::endl;
        return 1;
    }

    std::string line_in1;
    std::string line_in2;

    posit32_t pin1, pin2, pout;

    while (std::getline(file_in1, line_in1)) {
    // for (int i = 0; i < 10; i++)
    // {
    //     std::getline(file_in1, line_in1);
        std::getline(file_in2, line_in2);

            uint32_t value_in1 = std::bitset<32>(line_in1).to_ulong();
            uint32_t value_in2 = std::bitset<32>(line_in2).to_ulong();

            pin1 = castP32(value_in1);
            pin2 = castP32(value_in2);

            pout = p32_add(pin1,pin2);

            printBinary((uint64_t *)&pout.v, 32);
    }

    file_in1.close();
    file_in2.close();
    return 0;
}
// End of Arithmetic Computation using SoftPosit
```

Listing B.2: Arithmetic computation using Softposit

B.3 Quadratic Equation Solver

```
// Start of Quadratic Equation Solver
// GCC compiler: riscv32-corev-elf-gcc
#include <stdio.h>
#include <stdlib.h>

union PositBinary {
    float f;
    unsigned int i;
};

void solve_quadratic(union PositBinary a,
                     union PositBinary b,
                     union PositBinary c)
{
    union PositBinary two, four, discriminant, sqrt_discriminant, root1, root2;
    two.i = 0x48000000; // Posit representation of 2
    four.i = 0x50000000; // Posit representation of 4

    discriminant.f = b.f * b.f - four.f * a.f * c.f;
    if (discriminant.f < 0) {
        printf("No real roots.\n");
        return;
    }
    __asm__ ("fsqrt.s %0, %1" : "=f" (sqrt_discriminant.f) :
    "f" (discriminant.f));

    root1.f = (-b.f + sqrt_discriminant.f) / (two.f * a.f);
    root2.f = (-b.f - sqrt_discriminant.f) / (two.f * a.f);

    // Answer in Posit hex representation
    printf("Roots: %08x, %08x\n", root1.i, root2.i);
}

int main(int argc, char *argv[])
{
    union PositBinary a, b, c;
    // Roundabout way to assign Posit values
    a.i = 0x4c000000; // Posit representation of 3
    b.i = 0x6a400000; // Posit representation of 100
    c.i = 0x48000000; // Posit representation of 2

    solve_quadratic(a, b, c);
    //root1 = e6e07d55 = -0.0200120
    //root2 = 9bd5f945 = -33.313321

    return EXIT_SUCCESS;
}
// End of Quadratic Equatio Solver
```

Listing B.3: C code for solving a quadratic equation in Posit format

B.4 Monte Carlo PI estimation

```
// Start of PI estimation
#include <stdio.h>
#include <stdlib.h>
#include "randomValue.h"

union PositBinary {
    float f;
    unsigned int i;
};

int main(int argc, char *argv[])
{

  union PositBinary one, two, four;
  union PositBinary a_temp, b_temp, inside_circle, iter, result;

  inside_circle.i = 0;
  iter.i = 0x73e80000; // Posit representation of 1000
  one.i  = 0x40000000;  // Posit representation of 1
  two.i  = 0x48000000;  // Posit representation of 2
  four.i = 0x50000000; // Posit representation of 4

  unsigned iterations = 1000;
  for (unsigned int i = 0; i < iterations; i += 2) {
    a_temp.i = randomValues[i];
    b_temp.i = randomValues[i + 1];
    printf("Iteration= %d, inside_circle= %d\n", i, inside_circle);
    if (a_temp.f * a_temp.f + b_temp.f * b_temp.f <= one.f)
      inside_circle.f = inside_circle.f + one.f;
  }
    result.f =  four.f * inside_circle.f / (iter.f / two.f);
    printf("Estimated Pi: %08x\n", result.i);

  return EXIT_SUCCESS;
}
// End of PI estimation
```

Listing B.4: Monte Carlo PI Estimation

B.5 Custom Reciprocal

```
// Start of custom reciprocal algorithm
#include <stdio.h>
#include <stdlib.h>

union positBinary
{
  float f;
  unsigned int i;
};

union positBinary pi, ONE, P_100, Min;

int main(int argc, char *argv[])
{
  pi.i    = 0x4c90fdaa;  // Posit representation of Pi = 3.141592651605606
  ONE.i   = 0x40000000;  // Posit representation 1
  P_100.i = 0x6a400000;  // Posit representation 100
  Min.i   = 0x00000001;  // Posit representation of minimum value

  printf("---------------- Posit custom function start ----------------\n");

  union positBinary temp, temp_count;
  volatile union positBinary sum, count, mid_value;

  sum.i = 0x00000000; // Initial Sum
  temp.i = 0x40000000;
  count.i = 0x00000000; // Initial count
  temp_count.i = 0x78331115;

  int i = 0;
  for (i;;)
  {
    mid_value.f = ONE.f / (P_100.f * count.f + pi.f);
    sum.f += mid_value.f;

    if (mid_value.f == Min.f)
    {
      printf("[%d - %08x], %08x, %08x \n Value of mid_value reach minimum.
      Program finish\n", i, count.i, sum.i, mid_value.i);
      break;
    }
    else if (temp_count.f == count.f)
    {
      printf("[%d - %08x], %08x, %08x \n Value of n is equal to previous n.
      Program finish\n", i, temp.i, sum.i, mid_value.i);
      break;
    }
    else if (temp.f != sum.f)
    {
      // printf("[%d], %08x \n", i, sum.i);
      if (i % 5000 == 0)
      {
        printf("[%d - %08x], %08x, %08x \n", i, count.i, sum.i, mid_value.i);
      }
      else if (i == 8388608)
      {
        printf("[%d - %08x], %08x, %08x \n", i, count.i, sum.i, mid_value.i);
      }
      else if (i == 671089)
      {
        printf("[%d - %08x], %08x, %08x \n", i, count.i, sum.i, mid_value.i);
      }
      temp.f = sum.f;
      temp_count.f = count.f;
      i++;
      count.f += ONE.f;
    }
    else if (temp.f == sum.f)
    {
      printf("[%d - %08x], %08x, %08x \n value same finish calulation \n",
      i, count.i, sum.i, mid_value.i);
      break;
    }
    else
    {
      printf("error \n");
    }
  }

  i--; // as in the final loop, the value is the same.

  printf("\n reverse calculation \n");
  printf("initial [%d], %08x \n", i, sum.i);
  for (i; i >= 0;)
  {
    i--;
    count.f -= ONE.f;
    if (i % 5000 == 0)
    {
      printf("[%d - %08x], %08x, %08x \n", i, count.i, sum.i, mid_value.i);
    }
    mid_value.f = ONE.f / (P_100.f * count.f + pi.f);
    sum.f -= mid_value.f;
  }

  printf("Finish reverse calculation. \n");

  printf("----------------- Posit custom function end -----------------\n");

  return EXIT_SUCCESS;
}
// End of custom reciprocal algorithm
```

Listing B.5: Posit custom function